```
//获得一个数据库连接
Connection conn = DriverManager.getConnection(url,user,passw);
//获得一个 Statement 对象用于发送 SQL 语句
Statement stmt = conn. createStatement();
```

为了执行 Statement 对象的方法，需要把发送到数据库的 SQL 语句作为参数提供给 Statement 对象的方法，使用 Statement 对象执行语句。Statement 接口提供了如下所示的 3 个发送 SQL 语句的方法。

- ResultSet executeQuery(String sql)：用于产生单个结果集的语句，如 select 语句。
- ResultSet executeUpdate(String sql)：用于执行 insert、update 或 delete 语句等，返回值是一个整数，表示受影响的行数（即更新计数）。
- ResultSet execute(…)：运行语句，返回是否有结果集。

对于返回一个结果集的 executeQuery()方法，在检索完 ResultSet 对象的所有行时该语句完成。对于方法 executeUpdate()，当它执行时语句即完成。

4．操作结果集——ResultSet 接口

ResultSet 接口用于装载查询结果，并可以通过它的不同方法提取出查询结果。ResultSet 包含符合 SQL 语句条件的所有行，且它通过一套 GET 方法（这些 GET 方法可以访问当前行中的不同列）提供了对这些行中数据的访问。

ResultSet.next()方法将记录指针移动到 ResultSet 记录集的下一行，使之成为当前行。

注意： 结果集 ResultSet 是一张二维表，其中有查询所返回的列标题及相应的值，并且结果集的游标初始时是指向第一行的上一行，其实没有指向真实数据，所以在得到数据时，要首先调用语句 result.next()。

11.4　通过 JDBC 访问数据库

在体会了使用 JSP 和 JDBC 开发的一个简单数据库应用并了解了 JDBC 的相关知识后，就可以转入更加正式一点的学习过程了，下面将会介绍如何结合 JDBC 驱动程序的类型等相关知识开发实际的数据库应用程序。

JDBC 驱动程序可分为以下 4 个种类：JDBC-ODBC 桥加 ODBC 驱动程序、本地 API——部分用 Java 来编写的驱动程序、JDBC 网络纯 Java 驱动程序和本地协议纯 Java 驱动程序。

第 1、2 类驱动程序在直接的纯 Java 驱动程序还没有上市前将会作为过渡方案来使用；第 3、4 类驱动程序将成为从 JDBC 访问数据库的首选方法。在本书中介绍使用 JDBC-ODBC 桥加 ODBC 驱动程序和本地协议纯 Java 驱动程序，这也是比较常用的两种驱动程序，读者如果对另外两种驱动程序感兴趣可以参考相关的技术资料。

11.4.1　使用 JDBC-ODBC 桥连接数据库

JavaSoft（JDBC 的官方网站）桥产品利用 ODBC 驱动程序提供 JDBC 访问。使用这种

方式时，必须将 ODBC 二进制代码（许多情况下还包括数据库客户机代码）加载到使用该驱动程序的每个客户机上。因此，这种类型的驱动程序最适合于企业网，或者是用 Java 编写的三层结构的应用程序服务器代码。如图 11.9 所示是这种连接方法的结构。

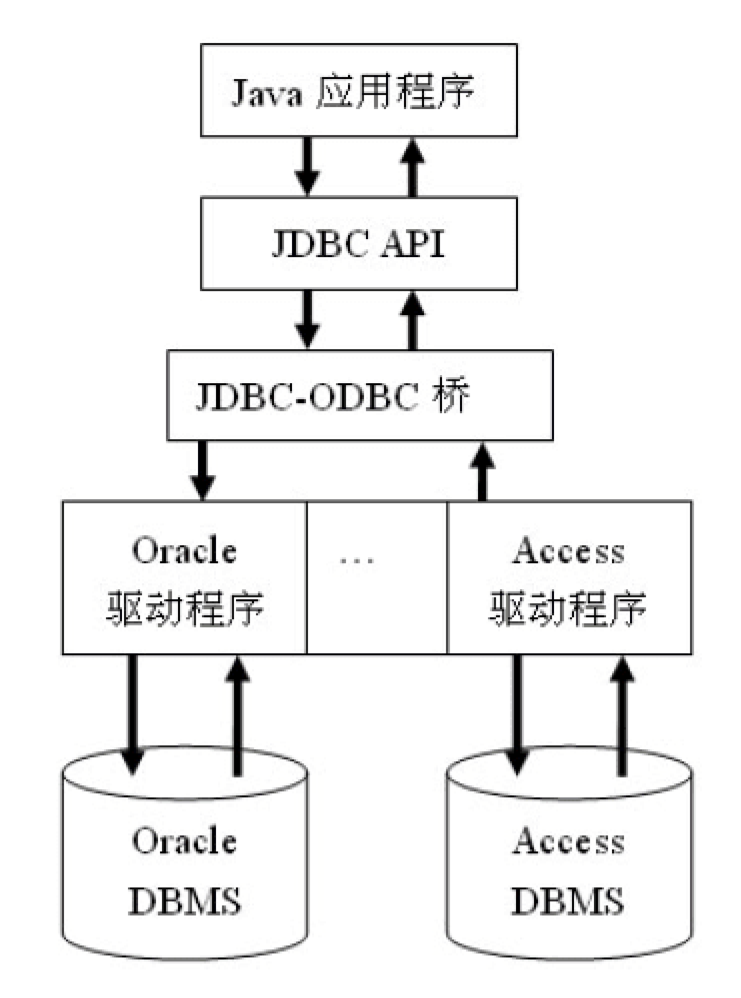


图 11.9　JDBC-ODBC 桥加 ODBC 驱动程序连接结构

下面介绍使用 JDBC-ODBC 桥加 ODBC 驱动程序实现数据库连接，由于这种方式暂时还不支持 MySQL 数据库（也可以安装组件实现支持 MySQL，不过不是一种很好的方式，这里不作介绍），所以在本例中不使用上面建立的 MySQL 数据库 jdbctestdb，而是使用了 Access 新建一个数据库。下面是使用这种驱动程序的基本步骤。

（1）建立数据库和数据表，数据库的名字是 jdbctestdb，数据表的名字是 animalInfo，具体建立数据库和数据表的方法请参考其他书籍。

（2）建立 ODBC 数据源，建立数据源（指建立 ODBC 数据源），数据源名称为 PetInfo，具体的建立方法请参考其他书籍。

注意： 建立 ODBC 数据源时，应该选择建立为系统 DSN。

（3）建立数据库连接，使用 JDBC 建立数据库的连接，首先要加载数据库驱动程序，实现此功能的语句如下：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

然后，使用特定的 URL 与数据库建立连接：

```
Drivermanger.getConnection(String url,String user,String password)
```

DriverManager 类用于处理驱动程序的载入并且对新的数据库连接提供支持。

（4）发送 SQL 语句，JDBC 提供了 Statement 类来发送 SQL 语句，因此要想执行 SQL 语句，必须要先建立 Statement 对象，Statement 对象由 Connection 类的 createStatement 方法创建，具体程序实现如下：

```
stmt=con.createStatement();
```

在建立 Statement 对象后，就可以用建立 Statement 对象发送 SQL 语句给数据库系统执行，执行返回的结果通常存放在一个 ResultSet 类的对象中。ResultSet 可以看作是一个表，这个表包含由 SQL 返回的列名和相应的值，ResultSet 对象中维持了一个指向当前行的指针，通过一系列的 getXXX 方法，可以检索当前行的各个字段，从而显示出来。下面是把上面的例子修改后的完整代码。

```
<!--   jdbcodbc.jsp -->
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*" errorPage
=" " %>
<html>
```

```
<head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
        <title>JDBC-ODBC 测试程序</title>
        <style type="text/css">
        <!--
        .style1 {
        font-size: 36px;
        font-family: Geneva, Arial, Helvetica, sans-serif;
        }
        -->
        </style>
</head>

<body>
<p>
   <%
          Connection conn=null;
          try
        {
          //加载数据库驱动程序
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();

        //与数据库建立连接，得到 Connection 的对象
          conn = DriverManager.getConnection("jdbc:odbc:PetInfo"," "," ");
        }catch(Exception e)
        {
                out.println(e);
        }

%>
</p>
<p align="center" class="style1">登记宠物信息</p>
<table width="700" border="1" align="center" bordercolor="#000000" bgcolor="#CCCCCC">
   <tr>
      <th scope="col">Animal ID</th>
      <th scope="col">Animal  类型</th>
      <th scope="col">Animal  名字</th>
      <th scope="col">Animal  主人</th>
      <th scope="col">Animal  生日</th>
      <th scope="col">Animal  描述信息</th>
   </tr>

<%
        try
        {
        //得到 Statement 的对象
          Statement stmt = conn.createStatement();
        //发送 SQL 语句，并得到执行结果
          ResultSet rs = stmt.executeQuery("select * from animalInfo");
```

```
        //处理返回结果
         while(rs.next()){
%>
   <tr bgcolor="#DDDDDD">
<td bgcolor="#DDDDDD"><div align="center"><%=rs.getString("ID")%></div></td>
<td bgcolor="#DDDDDD"><div align="center"><%=rs.getString("Ani_Type")%></div></td>
<td bgcolor="#DDDDDD"><div align="center"><%=rs.getString("Ani_Name")%></div></td>
<td bgcolor="#DDDDDD"><div align="center"><%=rs.getString("Ani_Owner")%></div></td>
<td bgcolor="#DDDDDD"><div align="center"><%=rs.getString("Ani_Birthday")%></div></td>
<td bgcolor="#DDDDDD"><div align="center"><%=rs.getString("Ani_Description")%></div></td>
</tr>
<%

}
//释放资源
rs.close();
stmt.close();
conn.close();
        }catch(Exception e)
        {
                out.println(e);
        }
%>
</table>
</body>
</html>
```

注意：（1）这份代码与第一节中代码的一个区别是获得数据库连接的方式改变了，还有在输出字符串时不需要进行编码方式的转换，因为中文版 Access 内部的默认编码方式是支持中文的。

（2）如果提示“javax.servlet.ServletException: [Microsoft][ODBC 驱动程序管理器] 未发现数据源名称并且未指定默认驱动程序”错误，请检查 ODBC 数据源是否被建立为系统 DSN。

这个程序运行后，效果如图 11.2 所示。

11.4.2　使用本地协议纯 Java 驱动程序连接数据库

本地协议纯 Java 驱动程序将 JDBC 调用直接转换为 DBMS 所使用的网络协议。这将允许从客户机上直接调用 DBMS 服务器，是 Intranet 访问的一个很实用的解决方法。由于许多这样的协议都是专用的，因此数据库提供者自身将是主要来源，有几家数据库厂商已开始做这些工作了。如图 11.10 所示是使用这种驱动程序连接数据库的结构。

使用本地协议纯 Java 驱动程序连接数据库的步骤和使用 JDBC-ODBC 桥加 ODBC 驱动程序连接数据库的过程基本是一致的，差别在于使用本地协议纯 Java 驱动程序连接数据库不需要建立 ODBC 数据源，也不能直接连接数据库文件（在使用 JDBC-ODBC 桥时是可以

实现的），事实上，在 11.1 节中介绍的例子就是使用这种驱动程序实现连接数据库的，下面介绍对这个例子进行扩展，使得它能对数据库进行查询、添加新的记录，并把访问数据库的代码从 JSP 文件中分离出来，提高 JSP 文件的简洁性。

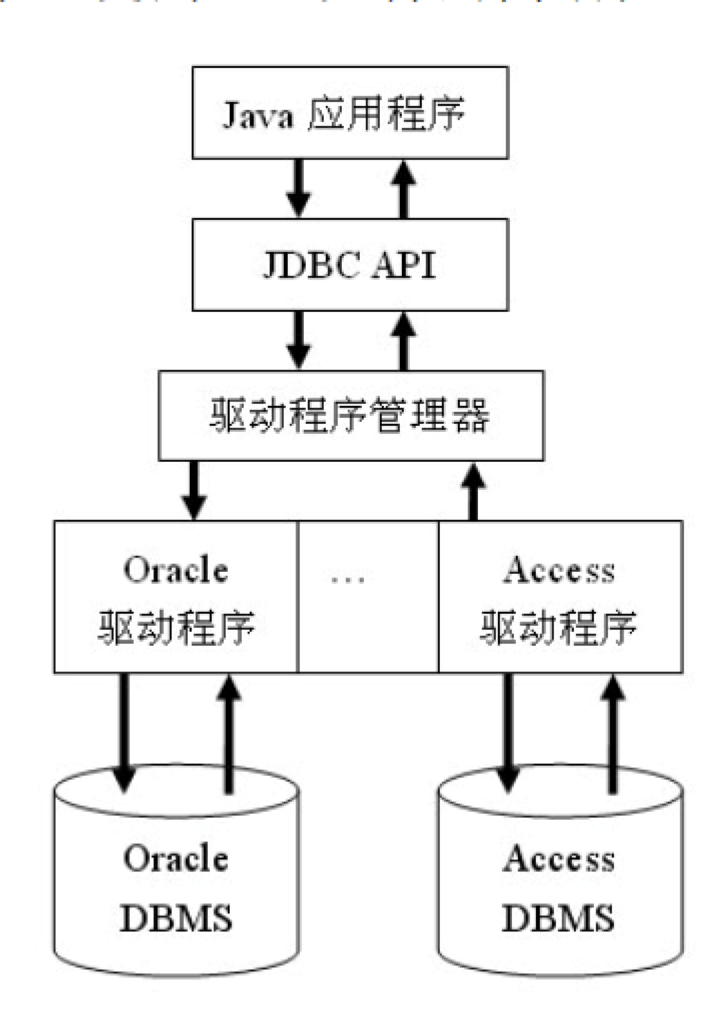


图 11.10　本地协议纯 Java 驱动程序

1．编写实现数据库操作的 JavaBeans

在本例中使用的 JavaBeans 要求具有数据库连接、数据库查询和更新数据库数据的功能，下面是 PetDBUtil.java 的源代码：

```
package cn.ac.ict;
import java.sql.*;
public class PetDBUtil {
     Connection conn=null;
     String url = "jdbc:mysql://localhost:3306/jdbctestdb";
     String user = "root";
     String passw = "ict";
    public PetDBUtil() {
        super();
    }

    //加载数据库驱动程序，并连接数据库
    public boolean connect(){
        try{
            //加载数据库驱动程序
             Class.forName("com.mysql.jdbc.Driver").newInstance();
            //与数据库建立连接，得到 Connection 的对象
             conn = DriverManager.getConnection(url,user,passw);
        }catch(Exception ex){
            //出错处理
            System.out.println("Drivers Load Error!");
            return false;
        }
        return true;
```

```
}

//执行数据库查询，传入的参数是使用的查询语句
public ResultSet query(String sql){
        ResultSet rs = null;
        Statement stmt;

        //检查是否已经连接到数据库
        if(conn==null){
                connect();
        }
        try {
                //得到 Statement 的对象
                stmt = conn.createStatement();
                //发送 SQL 语句，并得到执行结果
                rs = stmt.executeQuery(sql);
                return rs;
        } catch (SQLException e) {
                //  出错处理
                e.printStackTrace();
                return null;
        }
}

//执行数据更新，传入的参数是更新时使用的 SQL 语句
public boolean update(String sql){
        Statement stmt;
        int mcount;
        if(conn==null){
                connect();
        }
        try {
               //得到 Statement 的对象
                stmt = conn.createStatement();
                mcount = stmt.executeUpdate(sql);
                if(mcount<1){
                        return false;
                }
        } catch (SQLException e) {
                //  出错处理
                e.printStackTrace();
        }
        return true;
}

//回收资源，只有数据库连接 conn 是全局变量，只需要关闭 conn
public boolean close(){
        try {
                //释放资源
```

```
                conn.close();
            } catch (SQLException e) {
                // 出错处理
                e.printStackTrace();
            }
            conn=null;
            return true;
        }


}
```

在这个类中有 4 个方法，分别实现数据库连接、数据库查询、更新数据库数据和关闭数据库连接的功能，读者可以对照注释了解各个语句的作用。

2．编写调用 JavaBeans 的 JSP 文件

在本实例中使用了 6 个 JSP 文件，分别完成不同的功能，其中 pet.jsp 用于接受查询输入，其完整代码如下：

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage=" " %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>具有查询和更新功能的 JDBC 应用</title>
<style type="text/css">
<!--
.style1 {font-size: 24px}
-->
</style>
</head>
<body bgcolor="#FFFFFF">
<div align="center">
  <p>
  <span class="style1">  具有查询和更新功能的 JDBC 应用</span> </p>
  <table>
    <tr>
    <td><%@ include file="link.jsp"%></td>
  </tr>
  </table>
  <form action="petoperation.jsp" method="post" name="form1" target="_self">
    <div align="left">
      <p align="center">
        <label>输入要查询的条件
        <select name="type">
          <option value="ID" selected>ID</option>
          <option value="Ani_Type">宠物类型</option>
          <option value="Ani_Name">宠物昵名</option>
          <option value="Ani_Owner">宠物主人</option>
```

```
            <option value="Ani_Birthday">宠物生日</option>
          </select>
          <select name="comp">
            <option value="&gt;=" selected>&gt;=</option>
            <option value="&lt;=">&lt;=</option>
            <option value="=">=</option>
          </select>
          <input name="typevalue" type="text" value="2003-01-02" size="20" maxlength="20">
          </label>
        </p>
        <p align="center">
          <label>
          <input type="submit" name="Submit" value="提交">       
          <input name="reset" type="reset" id="reset" value="重置">
          </label>
        </p>
      </div>
    </form>
    <p>
    </p>
</div>
</body>
</html>
```

在 pet.jsp 的一个表单中声明表单被提交到 petoperation.jsp 页面，这是用于显示查询结果的一个页面，其完整代码如下：

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage=" " %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org
/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>具有查询和更新功能的 JDBC 应用</title>
    <jsp:useBean id="petUtil" scope="request" class="cn.ac.ict.PetDBUtil" />
<style type="text/css">
<!--
.style1 {font-size: 24px}
.style2 {font-size: 18px}
-->
</style>
</head>
<%
petUtil.connect();
String sel_type;
String sel_comp;
String sel_type_value;
String sql;
//获取请求参数
```

```
sel_type = request.getParameter("type");
sel_comp = request.getParameter("comp");
sel_type_value =request.getParameter("typevalue");
sql = "select * from animalinfo where "+sel_type+sel_comp+" ' "+sel_type_value+" ' ";
%>
<body>
<div align="center"><span class="style1">查询结果</span></div>
  <table>
    <tr>
    <td><%@ include file="link.jsp"%></td>
  </tr>
  </table>
<table width="700" border="1" align="center" bordercolor="#000000" bgcolor="#CCCCCC">
  <tr>
    <th scope="col">Animal ID</th>
    <th scope="col">Animal 类型</th>
    <th scope="col">Animal 名字</th>
    <th scope="col">Animal 主人</th>
    <th scope="col">Animal 生日</th>
    <th scope="col">Animal 描述信息</th>
    <th scope="col">修改</th>
  </tr>
  <p>
    <%
ResultSet rs = petUtil.query(sql);
if((rs!=null)&&(!rs.isLast())){
while(rs.next()){
%>
  <tr bgcolor="#DDDDDD">
    <td bgcolor="#DDDDDD"><div align="center"><%=rs.getString("ID")%></div></td>
    <td bgcolor="#DDDDDD"><div align="center"><%=new String(rs.getString("Ani_Type"). getBytes
("ISO8859-1"),"gb2312")%></div></td>
    <td bgcolor="#DDDDDD"><div align="center"><%=new String(rs.getString("Ani_Name"). getBytes
("ISO8859-1"),"gb2312")%></div></td>
    <td bgcolor="#DDDDDD"><div align="center"><%=new String(rs.getString("Ani_Owner"). getBytes
("ISO8859-1"),"gb2312")%></div></td>
    <td bgcolor="#DDDDDD"><div align="center"><%=new String(rs.getString("Ani_Birthday"). getBytes
("ISO8859-1"),"gb2312")%></div></td>
    <td bgcolor="#DDDDDD"><div align="center"><%=new String(rs.getString("Ani_Description").
getBytes("ISO8859-1"),"gb2312")%></div></td>
     <td bgcolor="#DDDDDD"><div align="center"><a href="modify.jsp?id=<%=rs.getString("ID")
%>">修改</a></div></td>
  </tr>
    <%}
     }%>

</body>
</html>
```

在显示查询结果的 petoperation.jsp 页面中每条记录都会有一个超链接用于修改记录，用

于接受修改数据的页面是 modify.jsp，其完整代码如下：

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage=" " %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR
/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
  <title>修改宠物记录</title>
<style type="text/css">
<!--
.style1 {font-size: 18px}
-->
</style>
</head>
  <!---     声明一个使用的 JavaBeans     ---->
<jsp:useBean id="petUtil" scope="request" class="cn.ac.ict.PetDBUtil" />
<%
String id =request.getParameter("id");
String sql = null;
ResultSet rs = null;
if(id!=null){
sql ="select * from animalinfo where ID=' "+id+" ' ";
//查询数据库获取结果
rs = petUtil.query(sql);
rs.next();
%>
<body>
<table width="700" border="0" align="center">
  <tr>
    <td><div align="center" class="style1">修改宠物记录
    </div></td>
  </tr>
    <tr>
    <td><%@ include file="link.jsp"%></td>
  </tr>

  <tr>
    <td height="135">
     <form target="_self" action="modify_confirm.jsp?id=<%=id%>" method="post">
     <table width="500" border="0" align="center">
       <tr bgcolor="#CCCCCC">
         <td><div align="center">宠物 ID</div></td>
         <td>
           <div align="left">
             <label><%=rs.getString("ID")%></label>
           </div></td>
       </tr>
       <tr bgcolor="#CCCCCC">
```

```
            <td> <div align="center">宠物类型 </div></td>
            <td>
              <div align="left">
                <input  name="Ani_Type"  type="text"  id="Ani_Type"  value="<%=new  String(rs.
getString("Ani_Type").getBytes("ISO8859-1"),"gb2312")%>">
              </div></td>
          </tr>
          <tr bgcolor="#CCCCCC">
            <td> <div align="center">宠物昵名 </div></td>
            <td>
              <div align="left">
                <input  name="Ani_Name"  type="text"  id="Ani_Name"  value="<%=new  String(rs.
getString("Ani_Name").getBytes("ISO8859-1"),"gb2312")%>">
              </div></td>
          </tr>
          <tr bgcolor="#CCCCCC">
            <td> <div align="center">宠物主人 </div></td>
            <td>
              <div align="left">
                <input  name="Ani_Owner"  type="text"  id="Ani_Owner"  value="<%=new  String(rs.
getString("Ani_Owner").getBytes("ISO8859-1"),"gb2312")%>">
              </div></td>
          </tr>
          <tr bgcolor="#CCCCCC">
            <td> <div align="center">宠物生日 </div></td>
            <td>
              <div align="left">
                <input  name="Ani_Birthday"  type="text"  id="Ani_Birthday"  value="<%=new  String
(rs.getString("Ani_Birthday").getBytes("ISO8859-1"),"gb2312")%>">
              </div></td>
          </tr>
          <tr bgcolor="#CCCCCC">
            <td> <div align="center">宠物描述信息 </div></td>
            <td>
              <div align="left">
                <input  name="Ani_Description"  type="text"  id="Ani_Description"   value="<%=new
String(rs.getString("Ani_Description").getBytes("ISO8859-1"),"gb2312")%>">
              </div></td>
          </tr>
           <tr bgcolor="#CCCCCC">
            <td> <div align="center">
              <input type="submit" name="Submit" value="提交">
            </div></td>
            <td><div align="center">
              <input name="reset" type="reset" id="reset" value="重置">
            </div></td>
          </tr>
        </table>
         </form>
```

```
        </td>
    </tr>
</table>
<%}%>
</body>
</html>
```

这个文件中表单的提交页面是 modify_confirm.jsp，在这个页面中完成将信息写到数据库中的任务，其完整代码如下：

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*" errorPage=" " %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>更新宠物记录</title>
</head>

<body>
        <jsp:useBean id="petUtil" scope="request" class="cn.ac.ict.PetDBUtil" />
<%
String id = request.getParameter("id");
String newid = request.getParameter("newid");
String type = request.getParameter("Ani_Type");
String name = request.getParameter("Ani_Name");
String owner = request.getParameter("Ani_Owner");
String birthday = request.getParameter("Ani_Birthday");
String description = request.getParameter("Ani_Description");
String sql=null;
petUtil.connect();
if(id!=null){
//构造 SQL 语句
sql = "update animalinfo set Ani_Type=' "+type+" ',Ani_Name=' "+name+" ',Ani_Owner=' "+owner+" ',
Ani_Birthday =' "+birthday+" ',Ani_Description=' "+description+" ' where ID=' "+id+" ' ";

}else if(newid!=null){
//构造 SQL 语句
sql = "insert into animalinfo values(' "+newid+" ',' "+type+" ',' "+name+" ',' "+owner+" ',' "+birthday+" ',
' "+ description+" ')";
}
//执行 SQL 语句，并判断是否执行成功
if(petUtil.update(sql)){
response.sendRedirect("pet.jsp");
}
out.print(sql);
%>
</body>
</html>
```

在这个页面中可以完成插入数据和修改数据的功能，另一个用于接受新加宠物信息的页面 add.jsp 的表单的提交目的也是这个页面，它根据不同的 ID 区分是修改数据还是新添加数据。接受新加宠物信息的页面 add.jsp 的完整代码如下：

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage=" " %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR
/html4/loose.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
    <title>添加宠物记录</title>
  <style type="text/css">
  <!--
  .style1 {font-size: 18px}
  -->
  </style>
</head>

<body>
<table width="700" border="0" align="center">
  <tr>
    <td><div align="center" class="style1">添加宠物记录
    </div></td>
  </tr>
  <tr>
    <td><%@ include file="link.jsp"%></td>
  </tr>
  <tr>
    <td height="135">
     <form target="_self" action="modify_confirm.jsp" method="post">
     <table width="500" border="0" align="center">
      <tr bgcolor="#CCCCCC">
        <td><div align="center">宠物 ID</div></td>
        <td>
          <div align="left">
            <input name="newid" type="text" id="newid">
          </div></td>
      </tr>
      <tr bgcolor="#CCCCCC">
        <td> <div align="center">宠物类型 </div></td>
        <td>
          <div align="left">
            <input name="Ani_Type" type="text" id="Ani_Type">
          </div></td>
      </tr>
      <tr bgcolor="#CCCCCC">
        <td> <div align="center">宠物昵名 </div></td>
        <td>
```

```
                <div align="left">
                    <input name="Ani_Name" type="text" id="Ani_Name">
                </div></td>
        </tr>
        <tr bgcolor="#CCCCCC">
            <td> <div align="center">宠物主人 </div></td>
            <td>
                <div align="left">
                    <input name="Ani_Owner" type="text" id="Ani_Owner">
                </div></td>
        </tr>
        <tr bgcolor="#CCCCCC">
            <td> <div align="center">宠物生日 </div></td>
            <td>
                <div align="left">
                    <input name="Ani_Birthday" type="text" id="Ani_Birthday">
                </div></td>
        </tr>
        <tr bgcolor="#CCCCCC">
            <td> <div align="center">宠物描述信息 </div></td>
            <td>
                <div align="left">
                    <input name="Ani_Description" type="text" id="Ani_Description">
                </div></td>
        </tr>
         <tr bgcolor="#CCCCCC">
            <td> <div align="center">
                <input type="submit" name="Submit" value="提交">
            </div></td>
            <td><div align="center">
                <input name="reset" type="reset" id="reset" value="重置">
            </div></td>
        </tr>
    </table>
     </form>
    </td>
  </tr>
</table>
</body>
</html>
```

另外一个用于包含的文件中包含了一些连接信息，很简单，就不在此列举了。

3．发布运行 Web 应用

所有需要的文件都编写完成后，这个 Web 应用的结构如图 11.11 所示。把 JDBCFirst 文件夹复制到<TOMCAT_HOME>/webapps 目录下，启动 Tomcat，然后在浏览器地址栏中输入如下地址：http://localhost:8080/JDBCFirst/pet.jsp，可以看到页面显示如图 11.12 所示。

单击【提交】按钮，显示查询结果，页面显示如图 11.13 所示。

如果要修改 Kitty 的信息，可以单击 Kitty 后面的修改超链接，把描述信息修改为“活泼可爱的小猫咪”。显示页面如图 11.14 所示。

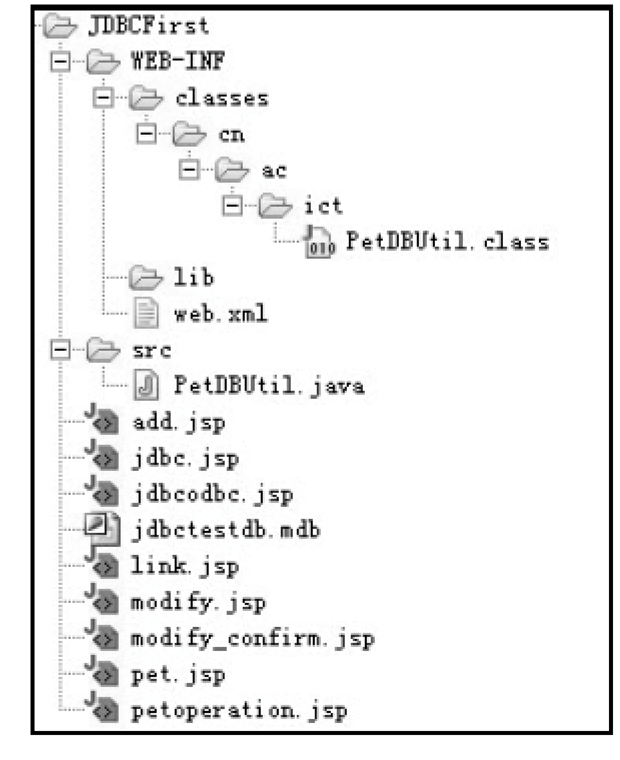


图 11.11　JDBC 应用结构



图 11.12　查询界面



图 11.13　查询结果



图 11.14　修改 Kitty 的信息

提交后，就可以通过再次查询查看修改的结果了。还有添加新记录的页面，在所有的显示页面上都提供了链接，单击【添加新的宠物记录】链接后，填入宠物的信息，显示如图 11.15 所示。

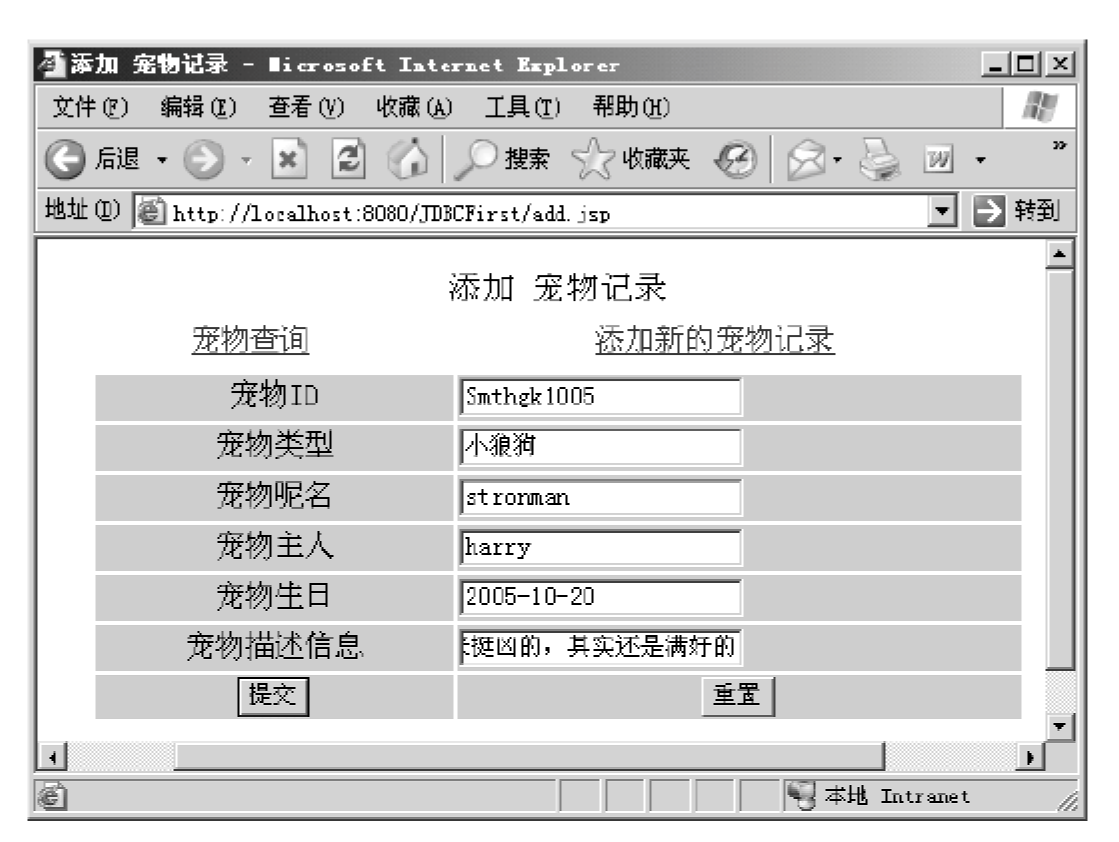


图 11.15　添加新的宠物记录

提交后，可以再查询查看修改结果，显示如图 11.16 所示。



图 11.16　数据更新结果

11.4.3　使用 PreparedStatement 接口发送 SQL 语句——数据录入例子

在 11.3 节中讲到 JDBC API 的一个比较重要的接口是 PreparedStatement，在上面的例子中已经可以完整地实现数据库的功能了，那么 PreparedStatement 接口是用来干什么的呢？PreparedStatement 接口继承于 Statement 接口，是经过扩展的一个用户发送 SQL 语句的接口，它与 Statement 在三个方面有所不同。

- ❑ PreparedStatement 实例包含已编译的 SQL 语句，就是使语句“准备好”的意思。包含于 PreparedStatement 对象中的 SQL 语句可含有一个或多个参数。参数的值在 SQL 语句创建时未被指定。相反地，该语句为每个参数保留一个问号（“?”）作为占位符。每个问号的实际值必须在该语句执行之前，通过适当的 setXXX 方法来提供。
- ❑ 由于 PreparedStatement 对象已预编译过，所以其执行速度要比 Statement 对象快。因此，多次执行的 SQL 语句创建为 PreparedStatement 对象，可以提高效率。
- ❑ 作为 Statement 的子类，PreparedStatement 继承了 Statement 的所有功能。另外它还添加了一整套方法，用于设置发送给数据库以取代参数占位符的值。同时，三种方法 execute、executeQuery 和 executeUpdate 已被更改以使之不再需要参数。这些方法的 Statement 形式（接受 SQL 语句参数的形式）不应该用于 PreparedStatement 对象。

下面介绍一个简单的例子来演示 PreparedStatement 接口的使用。在这个例子中借用了前面一直使用的数据库，用于向数据库中添加大量的宠物信息。

1．创建 PreparedStatement 对象

创建 PreparedStatement 对象与创建 Statement 是不同的，以下的代码段（其中 conn 是 Connection 对象）创建包含带两个参数占位符的 SQL 语句的 PreparedStatement 对象：

```
PreparedStatement pstmt = conn.prepareStatement("insert into animalinfo values(?,?,?,?,?,?)");
```

pstmt 对象包含语句" insert into animalinfo values(?,?,?,?,?,?)"，它已发送给 DBMS，并为执行作好了准备，但现在还不能得到任何执行结果，因为例子中在 SQL 语句中设置了多个参数，下面讲述如何传递这些参数。

2．传递参数

在执行 PreparedStatement 对象之前，必须设置每个“?”参数的值。这可通过调用一系

列 setXXX 方法来完成，其中 XXX 是与该参数相应的类型。例如，如果参数具有 Java 类型 Long，则使用的方法就是 setLong。setXXX 方法的第一个参数是要设置参数的序数位置，第二个参数是设置给该参数的值。例如，以下代码将第 4 个参数设为"jacky"，第 5 个参数设为 2005-10-21。

```
pstmt.setString(4,"jacky");
pstmt.setDate(5,new Date(105,9,21));
```

注意： java.sql.Date 日期的构造函数第一个参数是减去 1900 后的年数（2005-1900=105），第二个参数是月数（第一个月是 0，以后依次类推），第三个参数是日期数。

一旦设置了给定语句的参数值，就可用它多次执行该语句，直到调用 clearParameters 方法清除它为止。在连接的默认模式下（启用自动提交），当语句完成时将自动提交或回滚该语句（执行失败时）。

如果基本数据库和驱动程序在语句提交之后仍保持这些语句的打开状态，则同一个 PreparedStatement 可执行多次。如果没有这一功能，那么试图使用 PreparedStatement 对象代替 Statement 对象来提高性能是没有意义的。

下面是使用 PreparedStatement 对象的 JSP 文件的完整代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage=" " %>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
    <title>PreparedStatement 接口使用测试</title>
  <style type="text/css">
  <!--
  .style2 {font-size: 16px}
  -->
  </style>
</head>

<body>
<p align="left">
<%
      Connection conn=null;
      String url = "jdbc:mysql://localhost:3306/jdbctestdb";
      String user = "root";
      String passw = "ict";
      int id_name = 1010;
      int i=0;
      long   startime =0;
      long endtime =0;
      String id_prefix = "Smthgk";
      //加载数据库驱动程序
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      //获取数据库连接
      conn = DriverManager.getConnection(url,user,passw);
     //获取一个 PreparedStatement 对象
```

```
PreparedStatement pstmt = conn.prepareStatement("insert into animalinfo values(?,?,?,?,?,?)");
//下面设置各个参数的值
        pstmt.setString(2,new String("金丝鸟".getBytes("gb2312"),"ISO8859-1"));
        pstmt.setString(4,"jacky");
        pstmt.setDate(5,new Date(105,9,21));
        pstmt.setString(6,new String("大量养殖的鸟！".getBytes("gb2312"),"ISO8859-1"));
        startime = new java.util.Date().getTime();
        while(i<100){
        id_name+=1;
        pstmt.setString(1,id_prefix+id_name);
        pstmt.setString(3,new String(("佳佳"+id_name).getBytes("gb2312"),"ISO8859-1"));
        pstmt.execute();
        i++;
        }
        endtime = new java.util.Date().getTime();
        pstmt.clearParameters();
        pstmt.close();
        conn.close();
%>
<span class="style2">向数据库中写入 100 条记录总共用了<%=endtime-startime%>毫秒！</span>
</body>
</html>
```

程序运行后的效果如图 11.17 所示。

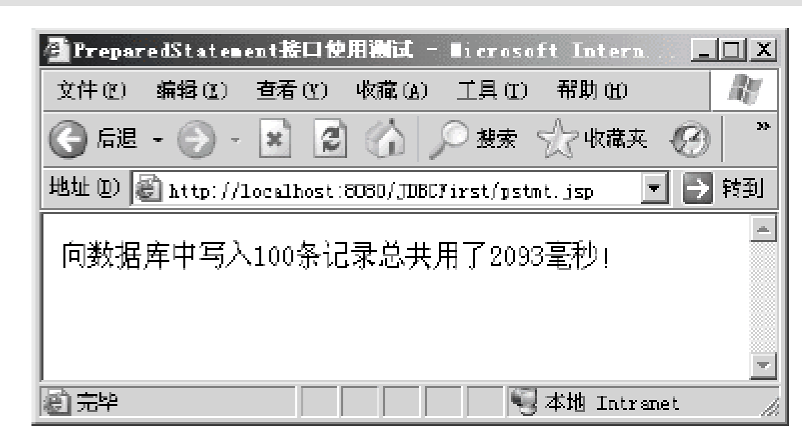


图 11.17　PreparedStatement 接口使用测试

3．参数中数据类型的一致性

setXXX 方法中的 XXX 是 Java 类型。它是一种隐含的 JDBC 类型（一般 SQL 类型），因为驱动程序将把 Java 类型映射为相应的 JDBC 类型，并将该 JDBC 类型发送给数据库。例如，在上面的例子中，把第 5 个参数设置为“2005-10-21”，例子中是使用 setDate 方式设置的，但事实上也是完全可以使用 setString 方法设置的。也就是说上面的 pstmt.setDate(5,new Date(105,9,21));是完全可以使用 pstmt.setString(5,"2005-10-21");来代替的。驱动程序将"2005-10-21"转换为 JDBC Date，然后发送给数据库，而例子中使用的 new Date(105,9,21)是 Java Date 类型的标准映射。程序员的责任是确保将每个参数的 Java 类型映射为与数据库所需的 JDBC 数据类型兼容的 JDBC 类型。

11.4.4　使用 JDBC 的数据库事务操作——银行转账

在实际应用中往往要求几句 SQL 语句同时执行或同时不执行，那么只使用上面提到的技术是不够的，而数据库事务就是用来解决这个难题的一个技术，下面介绍数据库事务以及如何通过 JDBC 使用数据库事务。

1．JDBC 的数据库事务概述

在 JDBC 的数据库操作中，一项事务是由一条或是多条表达式所组成的一个不可分割

的工作单元。通过提交 commit()或者回滚 rollback()来结束事务的操作。关于事务操作的方法都位于接口 Java.sql.Connection 中。

2．JDBC 的数据库事务特点

- 在 JDBC 中，事务操作默认是自动提交。也就是说，一条对数据库的更新表达式代表一项事务操作，操作成功后，系统将自动调用 commit()来提交，否则将调用 rollback()来回滚。
- 在 JDBC 中，可以通过调用 setAutoCommit(false)来禁止自动提交。之后就可以把多个数据库操作的表达式作为一个事务，在操作完成后调用 commit()来进行整体提交，倘若其中一个表达式操作失败，都不会执行到 commit()，并且将产生响应的异常。此时就可以在异常捕获时调用 rollback()进行回滚。这样做可以保持多次更新操作后相关数据的一致性。
- JDBC API 支持事务对数据库的加锁，并且提供了 5 种操作支持，两种加锁密度，其中 5 种支持是指如下几种操作支持。

```
static int TRANSACTION_NONE = 0;
```

禁止事务操作和加锁。

```
static int TRANSACTION_READ_UNCOMMITTED = 1;
```

允许脏数据读写（dirty reads）、重复读写（repeatable reads）和影象读写（phntom reads）。

```
static int TRANSACTION_READ_COMMITTED = 2;
```

禁止脏数据读写（dirty reads），允许重复读写（repeatable reads）和影象读写（phntom reads）。

```
static int TRANSACTION_REPEATABLE_READ = 4;
```

禁止脏数据读写（dirty reads）和重复读写（repeatable reads），允许影象读写（phntom reads）。

```
static int TRANSACTION_SERIALIZABLE = 8;
```

禁止脏数据读写（dirty reads）、重复读写（repeatable reads）和允许影象读写（phntom reads）。

两种密度是指上面的操作可以分为两类，其中最后一项为表加锁，其余 3、4 项为行加锁。对于上面提到的一些术语解释如下：

- 脏数据读写（dirty reads）：当一个事务修改了某一数据行的值而未提交时，另一事务读取了此行值。倘若前一事务发生了回滚，则后一事务将得到一个无效的值（脏数据）。
- 重复读写（repeatable reads）：当一个事务在读取某一数据行时，另一事务同时在修改此数据行。则前一事务在重复读取此行时将得到一个不一致的值。
- 影象读写（phntom reads）：当一个事务在某一表中进行数据查询时，另一事务恰好插入了满足查询条件的数据行。则前一事务在重复读取满足条件的值时，将得到一个额外的“影象”值。

JDBC 根据数据库提供的默认值来设置事务支持及其加锁，当然，也可以手工设置，使用 Connection 接口的如下方法设置。

```
setTransactionIsolation（TRANSACTION_READ_UNCOMMITTED）
```

可以使用 Connection 接口的如下方法查看数据库的当前设置。

```
getTransactionIsolation()
```

在进行手动设置时，数据库及其驱动程序必须支持相应的事务操作才行。

3．银行系统转账例子

数据库事务的一个典型应用就是银行系统的转账操作，当钱从一个账户取出后，就一定要存入对方账号，否则第一个账户中的钱就应该被放回原处。

首先需要建立一个数据表 bank_account 保存账户的信息，其字段的描述如表 11.3 所示。

表 11.3　数据表bank_account结构

	类　　型	是否可为空	是否为主键
ACC_ID	char(50)	否	是
username	char(20)	否	否
money	float	否	否

向其中添加两条数据记录，如表 11.4 所示。

表 11.4　数据表bank_account中的数据

ACC_ID	username	money
100223564587896	harry	53 423.1
100223564854566	mary	1000

现在要做的是 harry 要给 mary 转账 1000 块钱，实现的程序如下：

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage=" " %>
<html>
<head>
     <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
     <title>PreparedStatement 接口使用测试</title>
  <style type="text/css">
  <!--
  .style2 {font-size: 16px}
  .style3 {font-size: 18px}
  -->
  </style>
</head>

<body>
<p align="left">
<%
       Connection conn=null;
       String url = "jdbc:mysql://localhost:3306/jdbctestdb";
       String user = "root";
       String passw = "ict";
       String sql = null;
```

```
        //加载数据库驱动程序
        Class.forName("com.mysql.jdbc.Driver").newInstance();

        //获取数据库连接
        conn = DriverManager.getConnection(url,user,passw);
      //禁止自动提交，设置回滚点
        conn.setAutoCommit(false);
       Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery("select * from bank_account");
         %>
         <span class="style3">转账之前，账户的信息如下：</span>
<table width="500" border="0" align="center" bordercolor="#CCCCCC">
           <tr>
             <th scope="col">账号</th>
             <th scope="col">姓名</th>
             <th scope="col">剩余金额</th>
           </tr>
             <%while(rs.next()){%>
           <tr>
             <td><%=rs.getString("ACC_ID")%></td>
             <td><%=rs.getString("username")%></td>
             <td><%=rs.getFloat("money")%></td>
           </tr>
             <%}%>
        </table>
         <p align="left">            <%
          try{
         //数据库更新操作
         sql = "update bank_account set money=money-1000 where ACC_ID='100223564587896' ";
         stmt.executeUpdate(sql);
         sql = "update bank_account set money=money+1000 where ACC_ID='100223564854566' ";
         stmt.executeUpdate(sql);
//事务提交
         conn.commit();
         }catch(Exception ex){
             ex.printStackTrace();
            try {
                   //操作不成功则回滚
                    conn.rollback();
                  }catch(Exception e) {
                         e.printStackTrace();
              }
         }
//下面执行查询操作，查看事务的提交结果
      rs = stmt.executeQuery("select * from bank_account");
%>
         <span class="style3">转账之后，账户的信息如下：</span>
<table width="500" border="0" align="center" bordercolor="#CCCCCC">
           <tr>
             <th scope="col">账号</th>
```

```
            <th scope="col">姓名</th>
            <th scope="col">剩余金额</th>
        </tr>
          <%while(rs.next()){%>
        <tr>
            <td><%=rs.getString("ACC_ID")%></td>
            <td><%=rs.getString("username")%></td>
            <td><%=rs.getFloat("money")%></td>
        </tr>
          <%}%>
    </table>
     <%rs.close();
     stmt.close();
     conn.close();%>
    </body>
</html>
```

将这个文件复制到已经发布的 Web 应用中（如 JDBCFirst），然后访问这个应用，页面显示如图 11.18 所示。

这是转账成功时的效果，如果因为完整性约束的限制或者其他原因造成了两次更新操作中的任何一次失败都会使整个事务回滚，恢复到原来的状态，保障了储户的利益。

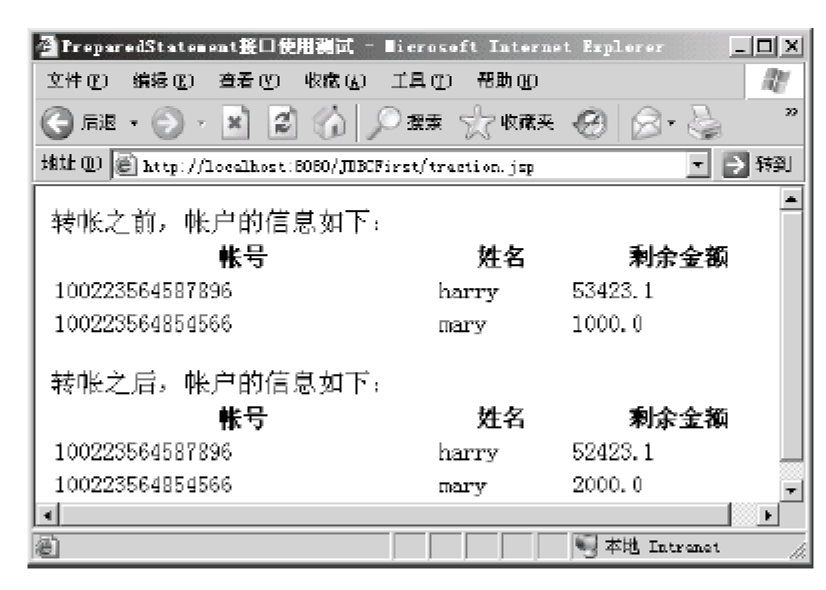


图 11.18　银行系统转账例子

11.5　JSP 与数据库连接池

在 Java 语言中，JDBC（Java Data Base Connection）是应用程序与数据库沟通的桥梁，一般来说，Java 应用程序访问数据库的过程（如图 11.19 所示）如下：

- ❑ 装载数据库驱动程序。
- ❑ 通过 JDBC 建立数据库连接。
- ❑ 访问数据库，执行 SQL 语句。
- ❑ 断开数据库连接。

但使用这种模式开发，存在很多问题。首先，系统要为每一次 Web 请求（例如下载一幅图片）建立一次数据库连接，对于一次或几次操作来讲，系统的开销还是比较小的，但对于 Web 程序来讲，即使在某一较短的时间段内，其操作请求数也远远不是一两次，而是数十上百次，在这种情况下，系统开销是相当大的。事实上，在一个基于数据库的 Web 系统中，建立数据库连接的操作将是系统中代价最大的操作之一。很多时候，一个网站速度瓶颈就在于此。

其次，使用传统的模式，系统必须去管理每一个连接，确保它们能被正确关闭，如果

出现程序异常而导致某些连接未能关闭，将导致数据库系统中的内存泄露，最终将不得不重新启动数据库。

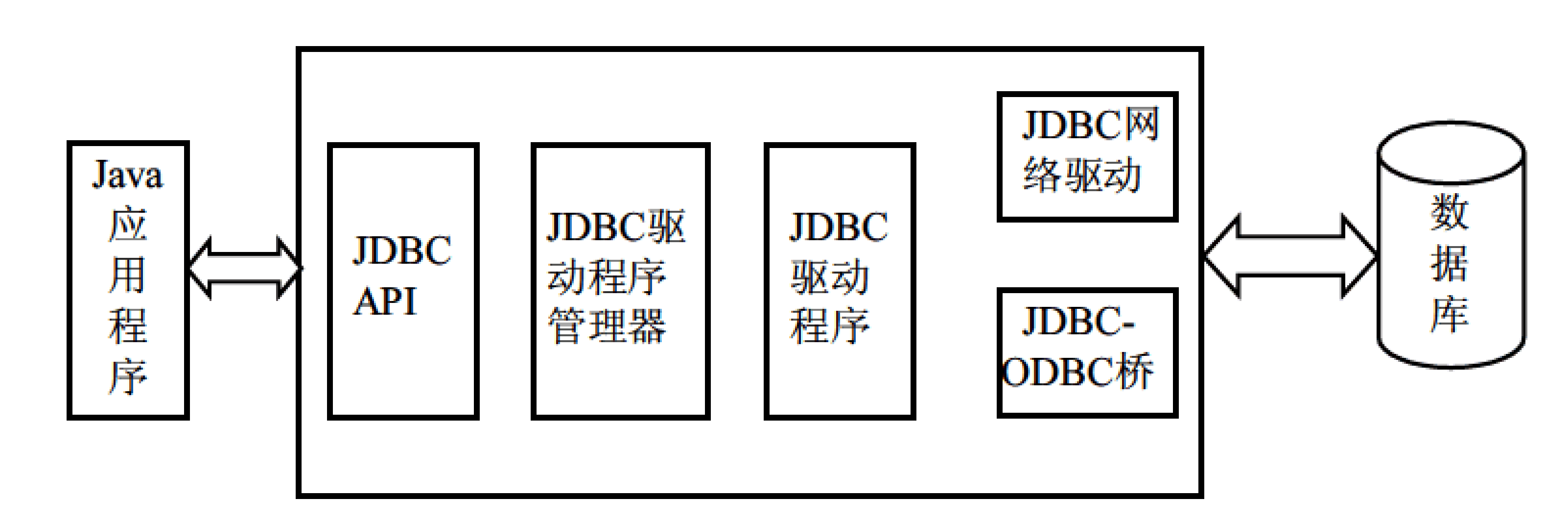


图 11.19　Java 应用程序访问数据库的过程

再次是安全性问题，由于程序代码中包含用户名和密码，其他人如果能得到字节码，可以通过反编译工具获得用户名和密码。还有就是代码的可移植性问题。如果希望连接的数据库名称或用户名有所更改，程序员需要修改源程序，然后把修改过的程序发送给用户。也就是说，软件无法脱离数据库独立存在。这样不仅会大大提高软件的成本，也不利于软件本身的发展。

基于传统模式存在的诸多问题，数据库连接池（Connection Pool）技术被提出来解决上述问题。数据库连接池模型应该包括一个连接池类（DBConnection Pool）和一个连接池管理类（DBConnection Pool Manager）。连接池类是对某一数据库所有连接的“缓冲池”，主要实现以下功能：

- 从连接池获取或创建可用连接。
- 使用完毕之后，把连接返还给连接池。
- 在系统关闭前，断开所有连接并释放连接占用的系统资源。
- 能够处理无效连接（原来登记为可用的连接，由于某种原因不再可用，如超时、通信问题），并能够限制连接池中的连接总数不低于某个预定值和不超过某个预定值。

连接池管理类是连接池类的包装类（Wrapper），符合单例模式，即系统中只能有一个连接池管理类的实例。其主要用于对多个连接池对象的管理，具有以下功能：

- 装载并注册特定数据库的 JDBC 驱动程序。
- 根据属性文件给定的信息，创建连接池对象。
- 为方便管理多个连接池对象，为每一个连接池对象取一个名字，实现连接池名字与其实例之间的映射。
- 跟踪客户使用连接情况，以便需要时关闭连接释放资源。

连接池管理类的引入主要是为了方便对多个连接池的使用和管理，如系统需要连接不同的数据库，或连接相同的数据库但由于安全性问题，需要不同的用户使用不同的名称和密码。

当程序中需要建立数据库连接时，只需从内存中取一个来用而不用新建。同样，使用完毕后，只需放回内存即可。而连接的建立、断开都由连接池自身来管理。同时，还可以通过设置连接池的参数来控制连接池中的连接数、每个连接的最大使用次数等。通过使用连接池，将大大提高程序效率，同时，可以通过其自身的管理机制来监视数据库连接的数量、使用情况等。

11.6　JSP 与数据源（DataSource）

11.6.1　数据源（DataSource）简介

数据源是在 JDBC 2.0 中引入的一个概念。在 JDBC 2.0 扩展包中定义了 javax.sql.DataSource 接口来描述这个概念。如果用户希望建立一个数据库连接，通过查询在 JNDI（Java Naming and Directory Interface，Java 名称和目录服务接口）服务中的数据源，可以从数据源中获取相应的数据库连接。这样用户就只需提供一个逻辑名称（Logic Name），而不是数据库登录的具体细节。

11.6.2　配置使用数据源

一般都是通过 JNDI 来使用数据源的，在本节中以在 Tomcat 中配置使用数据源来讲述如何在 JSP 技术中使用数据源。

1．准备 MySQL 数据库

在本例中使用“某实验室网站”的数据库名 space_web，读者可以参考相关 MySQL 的书籍建立数据库，由于本例只是验证使用数据源连接数据库，可以不用建立数据表。

2．配置数据源

下面介绍配置数据源的方法。数据源可以通过修改 server.xml 文件的 Context 元素配置数据源或者在 Context 片断中配置，它们的配置方法是一样的，当然数据源的配置也可以使用 Tomcat 提供的 Admin 系统配置管理应用程序实现。下面介绍使用修改 server.xml 文件 Context 元素配置数据源。

（1）打开 TOMCAT_HOME\conf\目录。

（2）找到并打开 server.xml 文件。

（3）在 Web 应用的虚拟目录中添加信息，如果所要配置的虚拟目录不存在，则在</Host>之前添加如下信息：

```
<?xml version='1.0' encoding='utf-8'?>
<!---   定义一个 Context 的属性信息，如 Web 应用的根目录和访问路径等    --->
<Context crossContext="true" debug="5" docBase="E:/PublishBook/lab" path="/lab" reloadable
="true" swallowOutput="true" workDir="work\Catalina\localhost\lab">
<!--
<Logger>是创建这个应用的 log 文件信息，其中 prefix 和 suffix 决定了日志文件的名字和扩展名，这
样设置后会在 TOMCAT_HOME/ logs/目录下生成 localhost_log.2005-05-04.txt 文件
-->
    <Logger className="org.apache.catalina.logger.FileLogger"
              prefix="localhost_lab_log." suffix=".txt"
              timestamp="true"/>
<!--
```

```
<Resource>被定义为该应用可以使用的资源，最主要的就是把一个 DataSource 与这个应用相关联了，并设定相关的参数-->
  <Resource name="jdbc/lab" type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/lab">
    <!-- 等待一个数据库连接有效的最长时间            -->
    <parameter>
      <name>maxWait</name>
      <value>5000</value>
    </parameter>
    <!--在连接池中活动数据库连接的最大数目,
     应该配置 MySQL 数据库，使得它可以满足所有应用的需要，设置为 0 表示没有任何限制-->
    <parameter>
      <name>maxActive</name>
      <value>4</value>
    </parameter>
    <!-- MySQL 数据库系统的用户名和密码-->
    <parameter>
      <name>username</name>
      <value>root</value>
    </parameter>      <parameter>
      <name>password</name>
      <value>ict</value>
    </parameter>
    <!-- JDBC 连接使用的 URL 地址    -->
    <parameter>
      <name>url</name>
<value>jdbc:mysql://localhost:3306/shopdb?autoReconnect=true</value>
    </parameter>
    <!-- 官方 MySQL 的 JDBC 驱动程序的类名 -->
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <!--在连接池中不活动数据库连接的最大数目，设置为-1 表示没有任何限制 -->
    <parameter>
      <name>maxIdle</name>
      <value>2</value>
    </parameter>

  </ResourceParams>
</Context>
```

注意： 实际使用时，要把所有的中文注释全部去掉，否则会无法正确加载 Web 应用。

3．安装驱动程序

将 MySQL 的驱动程序文件 mysql-connector-java-3.1.8-bin.jar（读者也可以使用 MySQL 其他版本的 JDBC 驱动程序）复制到 TOMCAT_HOME\common\lib 目录下，如果只把驱动程序放在 Web 应用的\WEB-INF\lib 目录下，Tomcat 中的数据源是无法正确加载驱动程序的。

4．编写测试文件

在本例中编写一个测试文件，它使用了上面配置的数据源。下面是测试文件

（datasourceconn.jsp）的代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*,java.
lang.*" errorPage=" " %>
<%
	String lang = "CH";
//下面这些代码是用于网站版本转换的，不是数据源的必要成分，但为了让它能够完全替换原来的
conn.jsp 文件，这里也列出来
	String lang_session = (String)session.getAttribute("lang");
	String lang_request = (String)request.getParameter("language");
	if(lang_request==null||lang_request.equals(" ")){
		if((lang_session!=null)&&(!lang_session.equals(" "))){
			lang = lang_session;
			session.setAttribute("lang",lang);
		}else{
			session.setAttribute("lang","CH");
		}
	}else{
			lang = lang_request;
			session.setAttribute("lang",lang);

	}
	//下面使用数据源获取连接
	 Connection conn=null;
	 try{
//获得初始上下文对象
		InitialContext ctx = new InitialContext();
//查找并获得一个数据源对象
		DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/lab");
//从数据库连接池中得到一个数据库连接
		conn = ds.getConnection();
	}catch(Exception ex){
		out.println("数据库驱动加载出现错误！ "+ex);
	}
%>
```

读者可以自己在数据库 space-web 中建立数据表并添加部分数据，进行各种数据库操作，更好地理解数据源的使用。

11.7　小　　结

在本章中介绍了关系数据库和 SQL 语言的基础知识、使用 JDBC 操作数据库的一些知识，了解了常用数据库的连接方法，掌握了 JDBC 的几个关键类和常用接口。在本章中的一个重点是学会使用 JDBC 连接数据库，并进行各种数据库的操作。

本章中介绍的很多知识对于初学者是很有用的，而且对于一些小型的 Web 应用都是很适合采用的技术。

第 12 章　JSP 与 Java Mail Web 应用

Java 提供的 Java Mail API，使得发送邮件变得非常简单。而且随着企业级应用的扩展，很多企业的信息管理系统中需要使用独立的邮件系统发送邮件，这样就使得 Java Mail Web 应用得到了广泛的应用。本章介绍如何建立 Java Mail Web 应用。

12.1　Java Mail 的简单实例

12.1.1　准备邮件服务器

要运行本章中介绍的例子，都需要有一个邮件服务器，读者使用的是 Magic winmail 试用版，并新建立一个用户名为 jmailapp，密码也为 jmailapp，所属的域名是 domain.com，具体其安装方法可以参考其相关文档。

12.1.2　编写程序

在本小节编写一个简单的程序，这个程序可以用来发送邮件，可以指定发送的接收者。源代码（JMailSend.java）如下：

```
package cn.ac.ict;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.URLName;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class JMailSend {
	public static void main (String args[]){
		String smtpHost = args[0];
		String from =args[1];
		String to = args[2];
		String userName = args[3];
		String password = args[4];
		SmtpAuth    auth = null;
```

```
            // 获取系统属性
            Properties props = System.getProperties();
            auth = new SmtpAuth();
            auth.setUserinfo(userName,password);
            // 设置邮件服务器相关信息
            props.put("mail.smtp.host", smtpHost);
            props.put("mail.smtp.auth", "true");
            props.put("mail.smtp.port", "25");
            props.put("mail.transport.protocol","smtp");
            props.put("mail.store.protocol","imap");
            props.put("mail.stmp.class","com.sun.mail.smtp.SMTPTrasport");
            props.put("mail.imap.class","com.sun.mail.imap.IMAPStore");

            // 得到一个会话
            Session session =
              Session.getDefaultInstance(props, auth);
            session.setPasswordAuthentication(new
URLName(smtpHost),auth.getPasswordAuthentication());

            // 定义一个消息
            MimeMessage message = new MimeMessage(session);
            try {
                  message.setFrom(new InternetAddress(from));
                  message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
                  message.setSubject("JMail Test Application");
                  message.setText("You JMail Application is successful!");

                  // 发送消息
                  Transport.send(message);
                  System.out.print("Send Successfully!");
             } catch (AddressException e) {
                   e.printStackTrace();
             } catch (MessagingException e) {
                   e.printStackTrace();
             }
         }
}
```

如果邮件服务器需要进行验证，就需要使用一个 Authenticator 的对象，提供认证信息。在本实例中是通过一个 Authenticator 的子类 SmtpAuth 的对象 auth 来实现的。如下：

```
auth = new SmtpAuth();
auth.setUserinfo(userName,password);
……
// 得到一个会话
Session session = Session.getDefaultInstance(props, auth);
session.setPasswordAuthentication(new URLName(smtpHost),auth.getPasswordAuthentication());
```

在得到一个会话后，在这个会话的基础上建立一个消息：

```
// 定义一个消息
MimeMessage message = new MimeMessage(session);
```

在对这个消息的一些内容进行设置后，就可以发送出去了。

12.1.3　运行程序并查看效果

上面的程序是一个普通的 Java 应用程序，可以在命令行下编译运行，但必须把需要的 JAR 文件 activation.jar 和 mail.jar 放到类路径中。

编译完成后，使用如下命令运行上面的程序：

```
java JMailSend localhost jmailapp@domain.com javamailuser@163.com jmailapp jmailapp
```

注意： 各个参数的含义见程序的分配。

如果运行成功后，会提示：Send Successfully!。使用 FoxMail 查看邮件，效果如图 12.1 所示。

可以看到，上面的代码虽然很短，但却很容易地发送了邮件，在后面的介绍中，会发现使用 Java 处理邮件都是很容易的。

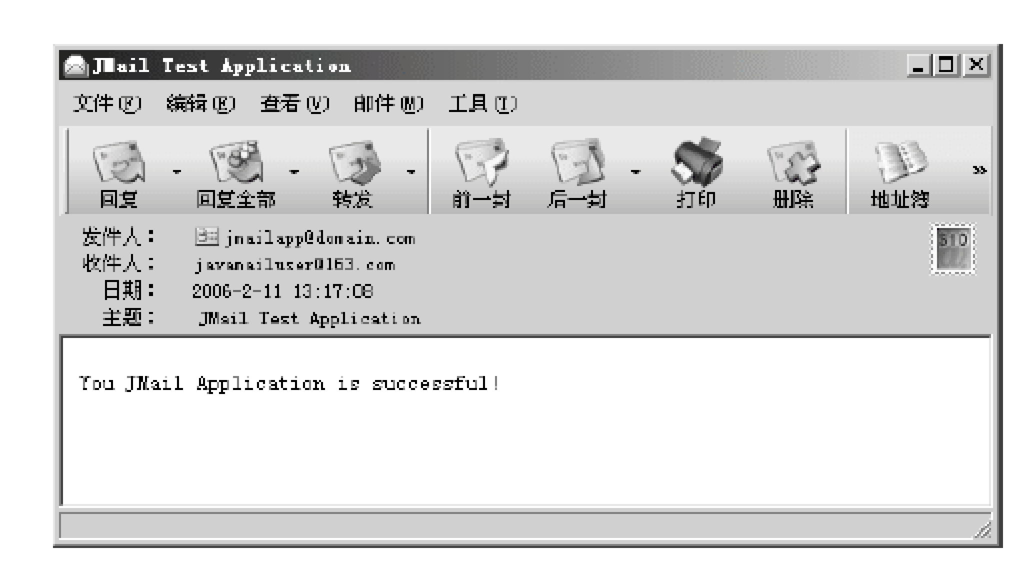


图 12.1　邮件发送成功

12.2　Java Mail API 简介

Java Mail API 是 Sun 开发的最新标准扩展 API 之一，它为 Java 应用程序开发者提供了独立于平台和协议的邮件/通信解决方案。Java Mail API 体系结构如图 12.2 所示。

Java Mail API 提供了一套 Internet 邮件系统模型的抽象类。它允许 Java 开发者在软件中使用 Java Mail API 提供的方法进行邮件的发送、接收和保存信息（可以不考虑信息的类型和协议）等，和 Java 的“一次写成，到处运行”的编程思想一致。Java Mail API 的核心内容包含在很少的几个主要的类中。本节将介绍这些主要的类和它们最常用的方法。

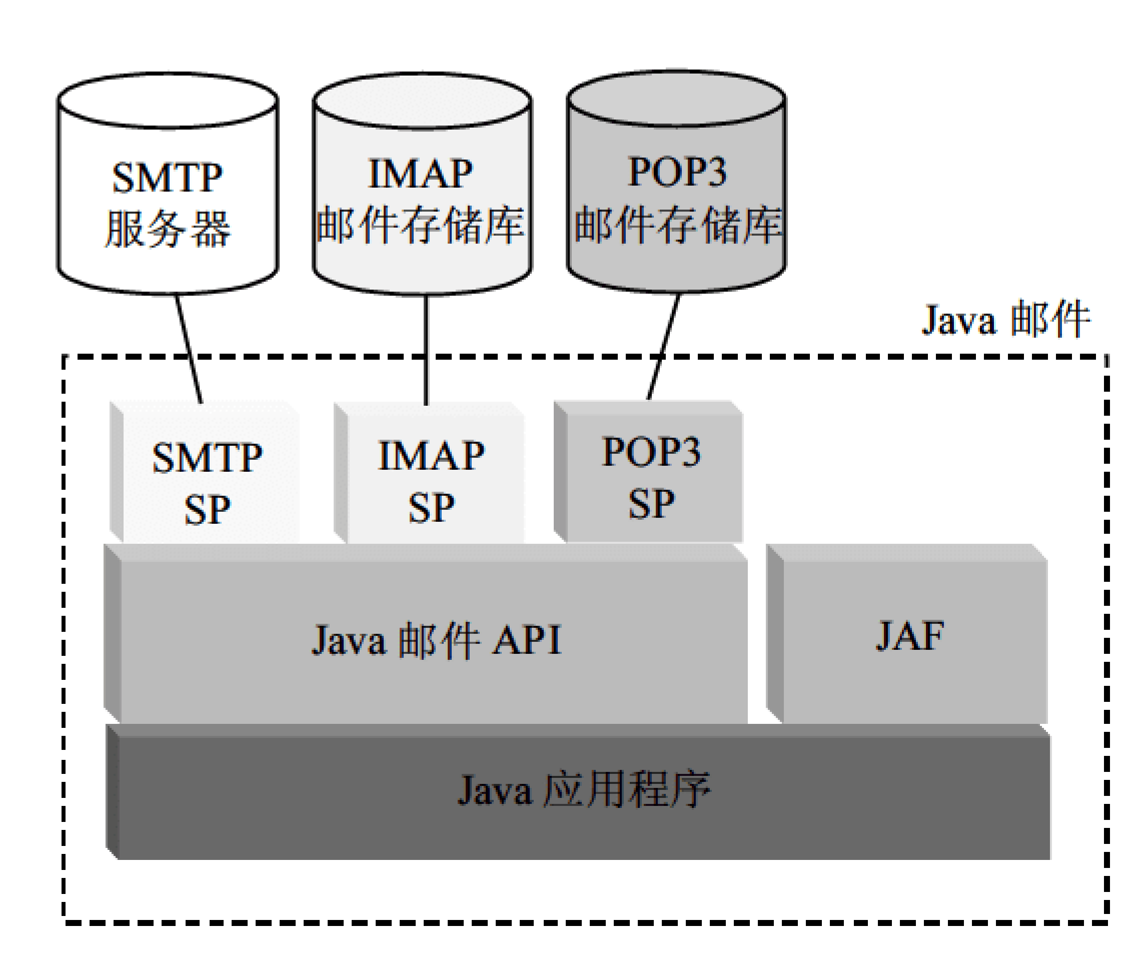


图 12.2　Java Mail API 体系结构图

1．javax.mail.Session

javax.mail.Session 是 Java Mail API 最高层入口类，它定义了一个基本的邮件会话。它最常用的方法是从 java.util.Properties 类的实例中获取相关信息，然后为不同邮件协议控制和装载 SPI（Service Provider Implementation）。例如，javax.mail.Store 是通过 Session 类获得的。

2．javax.mail.Store

javax.mail.Store 类实现特定邮件协议上的读、写、监视、查找等操作。通过 javax.mail.Store 类可以访问 javax.mail.Folder 类。

3．javax.mail.Transport

javax.mail.Transport 类也是由服务提供者提供的类，实现用特定协议发送消息/邮件。Transport 是一个抽象类，它提供了一个静态方法 send（Message）用来发送邮件。

4．javax.mail.Folder

javax.mail.Folder 类用于分级组织邮件，并提供按照 javax.mail.Message 格式访问 E-mail 的能力。

5．javax.mail.Message

javax.mail.Message 类模型化实际 E-mail 消息的所有细节，如标题、发送/接收地址、发送日期等。Message 是一个抽象类，它常用的子类为 javax.mail.internet.MimeMessage。MimeMessage 类是支持 MIME 类型电子邮件消息的类。

6．javax.mail.Address

Address 抽象类为消息中的地址建模，具体的子类提供详细的实现细节，它实现了 serializable 接口，最常用的子类是 javax.mail.internet.InternetAddress。

7．Java Mail API 与 JAF

需要注意的是 Java Mail API 实际上依赖于另外一个 Java 扩展 JAF，即 JavaBeans 活动框架（JavaBeans Activation Framework）。JAF 的目的在于统一处理不同数据格式的方法（不论数据格式为简单文本还是由图片、声音、视频甚至其他“活动”内容共同组成的复合文档）。在这个意义上，JAF 对 Java 的作用正如插件对 Web 浏览器的作用。

12.3　使用 Java Mail API 发送带附件的邮件应用

在本节中介绍如何使用 Java Mail API 发送带附件的邮件，并介绍一个能够发送带有 word 文件的应用程序。

12.3.1　编写程序

在本小节编写一个简单的程序，这个程序不但可以用来发送邮件，可以指定发送的接收者，实现 12.1 节中例子能实现的所有功能，而且可以发送带有附件的邮件。源代码

（EmailAttachBean.java）如下：

```
package cn.ac.ict;
import java.io.File;
import java.util.Properties;
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;

public class EmailAttachBean {

    public static void main(String[] args) {
        try {
        //在命令行下测试发送方法是否可以正常运行
        new EmailAttachBean().sendAttachMessage("127.0.0.1"," javamailuser@163.com",
"jmailapp@domain.com","Hello Attachment","e:\\An Introduction to the InfiniBand Architecture.
doc");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
      private void sendAttachMessage(String smtpServ, String to,
            String from,String subject,String attachment) throws Exception {
        Multipart multipart = null;
        MimeBodyPart   mbp1 = null;
        MimeBodyPart   mbp2 = null;

        Properties properties = System.getProperties( );
        //设置邮件服务器地址，这样默认的 Session 就可以使用它了
        properties.put("mail.smtp.host", smtpServ);
        Session session = Session.getDefaultInstance(properties);
        //建一个新的消息
        Message mailMsg = new MimeMessage(session);
        InternetAddress[] addresses = null;

        try {
            if (to != null) {
                // 如果收件人的地址不指定，就抛出异常
                addresses = InternetAddress.parse(to, false);
                mailMsg.setRecipients(Message.RecipientType.TO, addresses);
                } else {
                    throw new MessagingException("The mail message requires a 'To'
address.");
                    }
            //如果发件人的地址不指定，就抛出异常
            if (from != null) {
                mailMsg.setFrom(new InternetAddress(from));
                } else {
                throw new MessagingException("The mail message requires a valid 'From'
address.");
                }
```

```
                if (subject != null)
                        mailMsg.setSubject(subject);

                //邮件消息内容的类型是 'Multipart'
                //邮件消息的 MIME 类型是 'multipart/mixed'
                multipart = new MimeMultipart( );
                //下面介绍邮件消息的文本部分
                mbp1 = new MimeBodyPart( );
                String textPart = "Hello, just thought you'd be interested in this Word file.";
                //为文本部分消息内容创建一个 DataHandler 对象
                DataHandler data = new DataHandler(textPart, "text/plain");
                //设置文本 MimeBodyPart 的 DataHandler
                mbp1.setDataHandler(data);

                //把文本的 MimeBodyPart 加入到 Multipart 容器中
                multipart.addBodyPart( mbp1);

                //创建代表 word 附件的 MimeBodyPart
                mbp2 = new MimeBodyPart( );

                //创建一个指向文件的 DataHandler
                FileDataSource fds = new FileDataSource( new File(attachment));
                //确保附件被合适的 MIME 类型 application/msword 处理
                MimetypesFileTypeMap ftm = new MimetypesFileTypeMap( );

                //语法是：MIME 类型，然后空格，后面跟文件的扩展名
                ftm.addMimeTypes("application/msword doc DOC" );
                fds.setFileTypeMap(ftm);
                //使用刚创建的 FileDataSource 实例化 DataHandler
                DataHandler fileData = new DataHandler( fds );

              //让这个 MimeMultipart 包含刚才的 word 文件
                mbp2.setDataHandler(fileData);
                //为文件名指定字符集，否则数据虽然可以发到邮件中，但是无法识别出来
    mbp2.setFileName(MimeUtility.encodeWord(fds.getName(),"GB2312",null));
              //把包含了附件的 MimeMultipart 对象加到 Multipart 容器中
                multipart.addBodyPart( mbp2 );

              //最后把 MimeMessage 的内容设置为 Multipart 对象
                mailMsg.setContent( multipart );

              // 发送邮件
                Transport.send(mailMsg);
                } catch (Exception exc) {
                        throw exc;
                        }
                }

}
```

配置邮件服务器的地址、建立会话以及要发送的消息，这些内容和上一个例子都是一

样的。但发送带有附件的邮件与上面的例子就有所不同了。带有附件的邮件消息分为两部分：一部分是文本消息，另一部分是附件文件。这时，不论是文本消息还是附件都要分别放到一个 MimeBodyPart 对象中，而两个 MimeBodyPart 对象再放到一个 Multipart 对象中，构成一个完整的邮件消息，下面是构建文本消息 MimeBodyPart 对象的过程：

```
//下面介绍邮件消息的文本部分
mbp1 = new MimeBodyPart( );
String textPart = "Hello, just thought you'd be interested in this Word file.";
//为文本部分消息内容创建一个 DataHandler 对象
DataHandler data = new DataHandler(textPart, "text/plain");
//设置文本 MimeBodyPart 的 DataHandler
mbp1.setDataHandler(data);
//把文本的 MimeBodyPart 加入到 Multipart 容器中
multipart.addBodyPart( mbp1);
```

下面是构建 word 附件消息的 MimeBodyPart 对象的过程：

```
//创建代表 word 附件的 MimeBodyPart
mbp2 = new MimeBodyPart( );

//创建一个指向文件的 DataHandler
FileDataSource fds = new FileDataSource( new File(attachment));
//确保附件被合适的 MIME 类型 application/msword 处理
MimetypesFileTypeMap ftm = new MimetypesFileTypeMap( );

//语法是：MIME 类型，然后空格，岳面跟文件的扩展名
ftm.addMimeTypes("application/msword doc DOC" );
fds.setFileTypeMap(ftm);

//使用刚创建的 FileDataSource 实例化 DataHandler
DataHandler fileData = new DataHandler( fds );

//让这个 MimeMultipart 包含刚才的 word 文件
mbp2.setDataHandler(fileData);
//为文件名指定字符集，否则数据虽然可以发到邮件中，但是无法识别出来
mbp2.setFileName(MimeUtility.encodeWord(fds.getName(),"GB2312",null));
//把包含了附件的 MimeMultipart 对象加到 Multipart 容器中
multipart.addBodyPart( mbp2 );
```

最好把两个 MimeBodyPart 对象放到一个 Multipart 对象中后，将这个对象作为一个完整的邮件消息：

```
//最后，把 MimeMessage 的内容设置为 Multipart 对象
mailMsg.setContent( multipart );
```

最后就可以发送了。

12.3.2　运行程序并查看结果

上面的程序是一个普通的 Java 应用程序，可以在命令行下编译运行，但必须把需要的 JAR 文件 activation.jar 和 mail.jar 放到类路径中。

编译完成后，使用如下命令运行上面的程序：

```
java cn.ac.ict.EmailAttachBean
```

程序运行结束后，没有任何提示消息，使用 FoxMail 查看邮件，效果如图 12.3 所示。



图 12.3　发送带附件的应用

12.4　创建可发送附件的 Java Mail Web 应用

12.4.1　Java Mail Web 应用的程序分析

在这个 Java Mail Web 应用中，一个 Bean 实现了与邮件服务器的交互功能，还有两个辅助的类用于提供一些有用的功能；一个用于配置 Log4j，输出日志；其他的 JSP 文件用于实现显示部分的功能。各个程序的作用如表 12.1 所示。

表 12.1　Java Mail Web应用各个程序的作用

文　件　名	描　　述
MailUserInfoBean.java	管理邮件和邮件文件夹，保存客户信息
JMailUtil.java	工具集
Log4j.java	配置Log4j，输出日志
ConfigLog_ln.properties	Log4j的配置文件
connect.jsp	根据客户信息登录邮件服务器
listall.jsp	管理邮箱邮件夹
listone.jsp	管理邮件夹里的邮件
showmail.jsp	显示一封邮件，并对该邮件进行管理
write.jsp	写新邮件以及提交邮件
login.jsp	客户登录界面

客户访问 Web 应用的基本流程可以用图 12.4 来说明。

12.4.2　邮件账户管理

MailUserInfoBean.java 是 Java Mail Web 应用的主要实现类，在这个程序中提供了保存客户账户和会话的信息，它是一个具有在 Session 范围内有效的 JavaBeans，它的一些属性

用于连接邮件服务器或者保存当前的会话信息。它的属性如下：

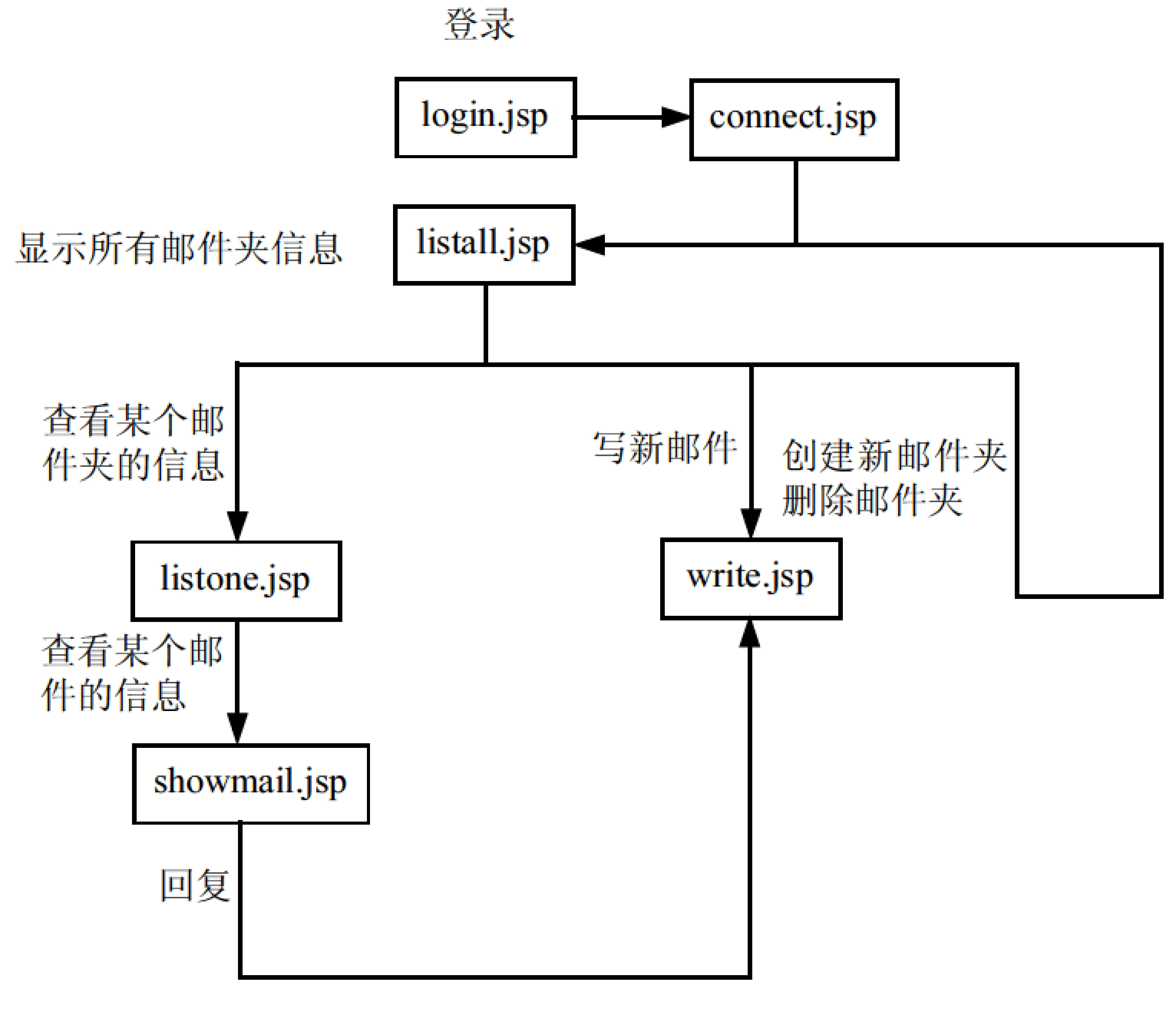


图 12.4　Java Mail Web 应用访问流程

```
//客户连接邮件服务器使用的 URL
    URLName urlName;
//客户当前的会话
    Session mailSession;
//客户使用的 Store
    Store store;
//客户当前访问的文件夹
    Folder currentFolder;
//客户当前访问的邮件
    Message currentMsg;
```

在这个类中分别提供了对应属性的 getter 和 setter 方法。

MailUserInfoBean 类提供了很多管理邮件的方法，如表 12.2 所示。后面将选取几个有代表性的方法讲述其工作过程。

表 12.2　管理邮件的方法

方　法　名	描　　述
deleteMessage	从指定的邮件夹中删除指定的邮件
createMessage	根据相关信息构造一个MIMEMessage
sendMessage	发送邮件
saveMessage	把邮件保存到草稿箱中
moveMessge	把邮件从一个文件夹移动到另外一个邮件夹
createTextMessage	构造一个没有附件的文本邮件消息
createAttachMessage	构造一个能带附件的文本邮件消息

删除邮件时要注意，Java Mail API 没有用于直接删除邮件的方法。要删除一个邮件，

首先把邮件的 DELETED 标志设置为 true，然后调用所在邮件夹的 expunge 方法，把这个邮件夹里所有 DELETED 标志为 true 的邮件删除。而从 Trash 中删除邮件和从其他邮件夹删除邮件是不一样的。如果从非 Trash 邮件夹删除邮件，需要把邮件复制到 Trash 邮件夹，然后从原来的邮件夹中删除邮件，而从 Trash 邮件夹中删除邮件就不需要复制了，而只要把邮件删除就可以了。下面是删除指定数目邮件的实现方法：

```
public int deleteMessage(int delArray[],Folder f){
  try{
        for(int i=0;i<delArray.length;i++){
//获取所有需要被删除的邮件
              if(delArray[i]==0) continue;
              Message delMsg = f.getMessage(i+1);
              if(f.getName().equals("Trash")){
                    Message[] m=new Message[1];
                    m[0] = delMsg;
     //把邮件复制到垃圾桶
        Folder Trash=store.getFolder("Trash");
                     f.copyMessages(m,Trash);
     //把邮件设置为已删除
                     delMsg.setFlag(Flags.Flag.DELETED, true);
              }else{
                    delMsg.setFlag(Flags.Flag.DELETED, true);
              }
        }
        f.expunge();
  }catch(Exception e){
        e.printStackTrace();
        Log4j.logger.debug("删除邮件失败！ "+e);
        return JMailUtil.FAILED;
  }
  Log4j.logger.info("邮件被成功删除!");
  return JMailUtil.SUCCESS;
}
```

发送邮件时需要注意，不但要把邮件发送到收件人的邮箱中，还需要把发送的邮件保存到自己的发件箱中。而把邮件保存到草稿箱中的操作和把邮件保存到发件箱类似。下面是发送邮件的代码，相似操作可以类推得到。

```
public int sendMessage(Message msg){
  try {
            //发送邮件
              Transport.send(msg);
            //把邮件保存到 SendBox
              Folder f=store.getFolder("SendBox");
              if(!f.isOpen())
                    f.open(Folder.READ_WRITE);
              Message m[]=new Message[1];
            m[0]=msg;
```

```
                    f.appendMessages(m);
                } catch (MessagingException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                    Log4j.logger.debug("发送邮件失败!"+e1);
                }
                Log4j.logger.info("发送邮件成功!");
            return JMailUtil.SUCCESS;
        }
```

把一个邮件从一个文件夹复制到另外一个文件夹的过程是先把邮件复制到另外一个邮件夹，然后把原来邮件夹里的邮件删除，相关操作在前面的代码中有所涉及，读者也可参考光盘上的源文件。

构建邮件消息是本应用的一个重点，不过它是在上面两个应用的基础上修改过来的，这里就不详细介绍了。

12.4.3　包含文件

link.jsp 是定义了常用链接的文件，在这个文件中定义了常用的如查看新邮件、写信、返回主目录以及退出等功能链接，方便用户浏览。

```
<%@ page contentType="text/html; charset=GB2312"%>
<table width="55%" border=0 align =center><tr>
<td><a href="listone.jsp?folder=INBOX">查看新邮件</a></td>
<td><a href="listall.jsp">主目录</a></td>
<td><a href="write.jsp">写新邮件</a></td>
<td><a href="logout.jsp">退出</a></td>
</tr>
</table>
```

效果如图 12.5 所示。

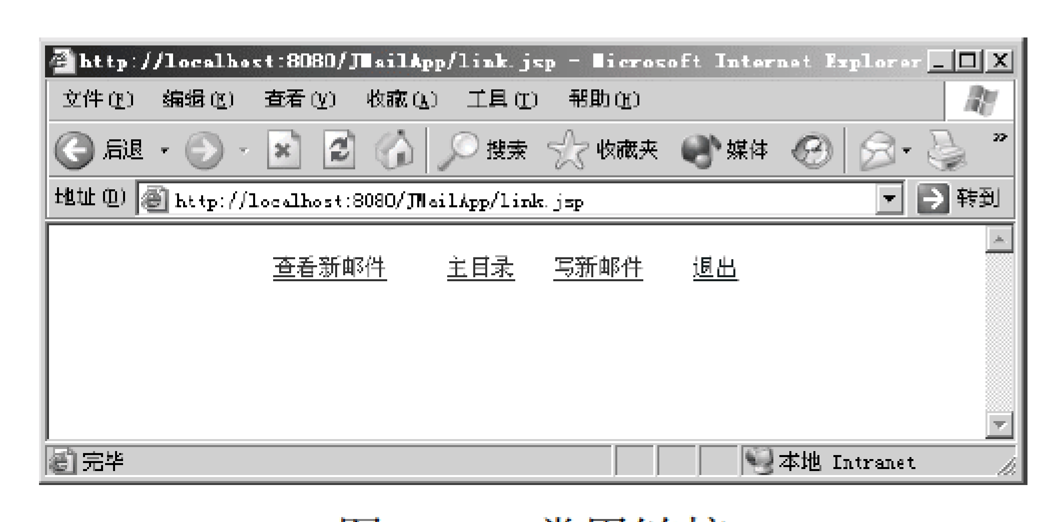


图 12.5　常用链接

12.4.4　登录邮件服务器

Login.jsp 是用来提供登录邮件服务器的页面。在本例中要登录本机安装的邮件服务器，可以按照如下格式填写（如图 12.6 所示），要确保安装好了邮件服务器，并建立一个账户 jmailapp，密码 jmailapp。



图 12.6　登录邮件服务器页面

用于提示登录的页面 Login.jsp 的源代码如下：

```
<%@ page contentType="text/html; charset=GB2312"%>
<html><head><title>JavaMail</title></head>
<body >
    <p><center><font face="Arial,Helvetica" font size=+3><b>欢迎使用 Java Mail Web 应用</b>
</font></center></p>
<center><img src="images\signature.gif"></center>

<%
  String loginfail=(String)request.getAttribute("loginfail");
  //判断用户是否登录失败
  if(loginfail!=null && loginfail.equals("true")){
%>
<center><p><font color="red">登录失败，请确认用户名和密码后再重试！ </font></p></center>
<%
}
%>

<form action="connect.jsp" method="post" >
<center>
<table   >
<tr>
    <td >用户名:</td>
    <td ><input type="text" name="username" size="25" value="jmailapp"></td>
</tr>
<tr>
    <td >密码:</td>
    <td ><input type="password" name="password" size="25" value="jmailapp"></td>
</tr>
</table>

</center>
<center><br>
    <input type="reset" name="Reset" value="重置">
    <input type="submit" value="提交">
</center>
</form>
```

```
</body>
</html>
```

login.jsp 的提交页面是 connect.jsp，实现连接的工作是由 connect.jsp 来完成的。它主要通过 MailUserInfoBean 的 connect 方法实现连接的操作，connect 方法是实现用户登录的主体部分。它在 MailUserInfoBean 中的方法定义如下：

```
    public int connect(String host,String user,String pass){
             Properties prop=null;
            prop = System.getProperties();
            prop.put("mail.transport.protocol", "smtp");
            prop.put("mail.store.protocol", "imap");
            prop.put("mail.smtp.class", "com.sun.mail.smtp.SMTPTransport");
            prop.put("mail.imap.class", "com.sun.mail.imap.IMAPStore");
            prop.put("mail.smtp.host", "localhost");
            this.mailSession = Session.getDefaultInstance(prop, null);

            try {
                    //  获取一个 Store 对象
                        this.store = this.mailSession.getStore("imap");
//使用 Store 对象连接邮件服务器
                        this.store.connect(host,user,pass);
                        Log4j.logger.info("The Store is connected!");
                } catch (NoSuchProviderException e) {
                    e.printStackTrace();
                    Log4j.logger.debug("Get a Store error!");
                    Log4j.logger.debug(e);
                    return JMailUtil.FAILED;
                } catch (MessagingException e) {
                    Log4j.logger.debug("Get a Store error!");
                    e.printStackTrace();
                    Log4j.logger.debug(e);
                    return JMailUtil.FAILED;
                }
//返回操作成功信息
                return JMailUtil.SUCCESS;
    }
```

虽然 connect 方法提供了进行连接的大部分功能，但还不能完成基本的初始化或保证连接的作用。在 connect.jsp 中首先判断客户是否获得连接，如果没有连接，就要重新尝试连接，连接成功后，做一点简单的初始化，然后把页面转到 listall.jsp（主页面）。connect.jsp 的源代码如下：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import ="cn.ac.ict.JavaMail.*,javax.mail.Store"%>
<%@ page import ="javax.mail.Folder"%>
<%@ page import ="javax.mail.URLName"%>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
```

```
<%@ page import="javax.activation.*" %>
<jsp:useBean id="userInfo" scope="session" class="cn.ac.ict.JavaMail.MailUserInfoBean" />
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>Lomboz JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
    String hostname = "localhost";//request.getParameter("hostname");
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    //连接邮件服务器
    if(userInfo.connect(hostname,username,password)!=JMailUtil.SUCCESS){
    request.setAttribute("loginfail","true");
    out.println("get Connection error occur!");
 %>
 <%}
     Store store = userInfo.getStore();
     //判断是否已经连接到邮件服务器
     if(!store.isConnected()){
          store.connect(hostname,username,password);
          out.print("OK!"+store.isConnected());
     }
   Folder folder=store.getFolder("Trash");
   if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

   folder=store.getFolder("SendBox");
   if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

   folder=store.getFolder("Draft");
   if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

   folder.open(Folder.READ_WRITE);

   //构建使用的 URL
   URLName url = new URLName("imap",hostname, -1, "inbox", username, password);
   userInfo.setURLName(url);
   out.println("OK!");
 %>
 <jsp:forward page="listall.jsp" />

</body>
</html>
```

12.4.5　管理邮件夹中的邮件

在 listall.jsp 页面中单击任何一个邮件夹的名字可以进入管理邮件的页面 listone.jsp，在

这个页面中可以显示用户邮件夹中邮件的数目、未读邮件的数目以及所有邮件的列表。对邮件的管理功能是可以删除指定的邮件，页面如图 12.7 所示。

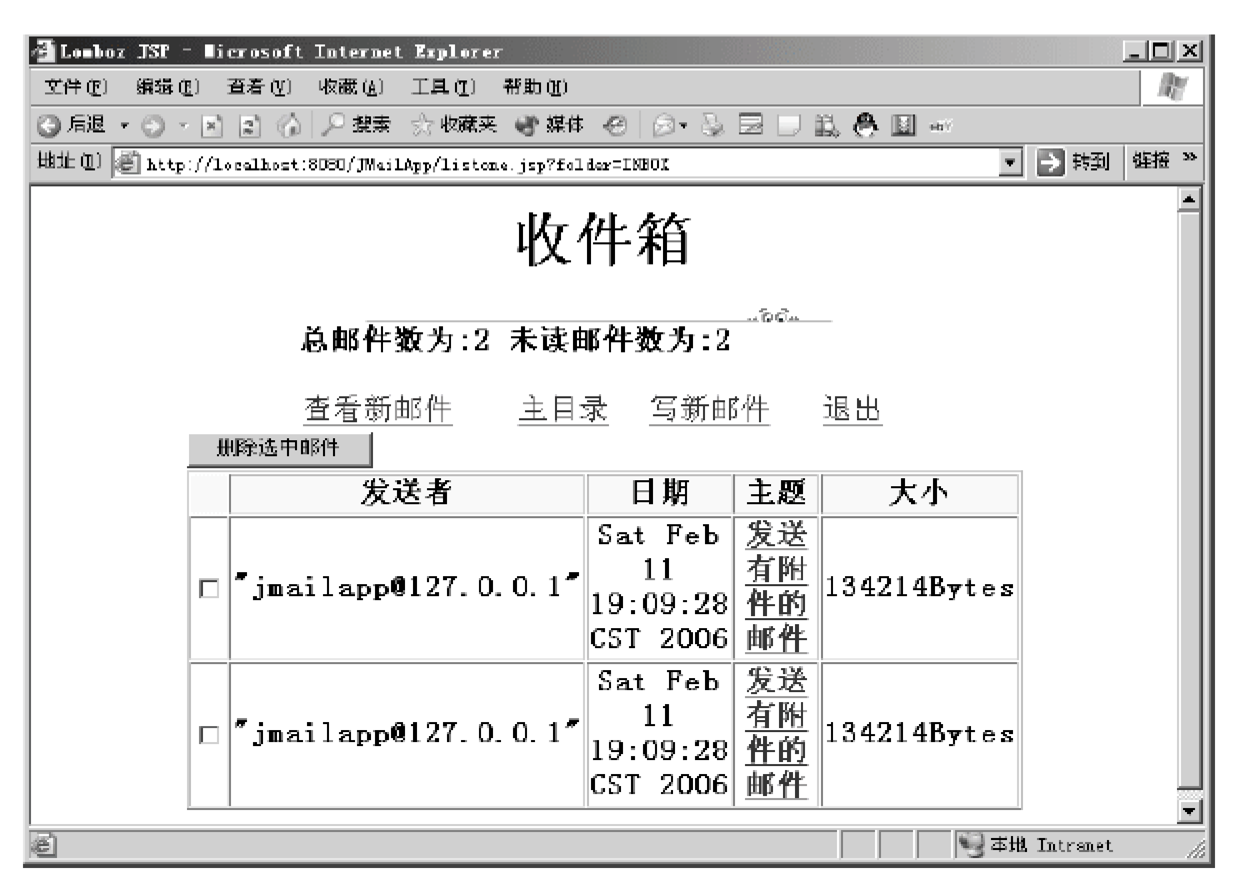


图 12.7　管理邮件夹中的邮件

通过 Message 类的 isSet（Flags.Flag.SEEN）可以判断这封邮件是否被阅读过，这是一个很有用的方法。更详细的使用技巧可以参考下面 listone.jsp 的源代码：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import ="cn.ac.ict.JavaMail.*,javax.mail.Store"%>
<%@ page import ="javax.mail.Folder"%>
<%@ page import ="javax.mail.URLName"%>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>
<jsp:useBean id="userInfo" scope="session" class="cn.ac.ict.JavaMail.MailUserInfoBean" />
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
     <title>显示邮件信息</title>
</head>

<body bgcolor="#FFFFFF">

<%
String folderName=request.getParameter("folder");
    Folder f=null;
    if(folderName!=null){
    //如果用户指定了邮件夹
        f=userInfo.getStore().getFolder(folderName);
        userInfo.setCurrentFolder(f);
    }else{
      //如果用户没有指定邮件夹，就使用当前的邮件夹
     f=userInfo.getCurrentFolder();
      folderName=f.getName();
    }
```

```
//如果邮件夹没有打开，就打开这个邮件夹
   if(!f.isOpen())f.open(Folder.READ_WRITE);
  int msgCount = f.getMessageCount();
  int unReadCount = f.getUnreadMessageCount();
    //删除邮件消息
  int arrayOpt[]=new int[msgCount];
  for(int i=1;i<=msgCount;i++){
   String optS=request.getParameter("delIndex"+i);
   if(optS!=null) arrayOpt[i-1]=1;
  }
     userInfo.deleteMessage(arrayOpt,f);
       //更新邮件数
   if(f.isOpen())f.close(true);
   f.open(Folder.READ_WRITE);
  msgCount = f.getMessageCount();
  unReadCount = f.getUnreadMessageCount();

%>
<center><font size="+3"><b><%=JMailUtil.getChinese(folderName)%></b></font></center><p>
<center><img src="images\signature.gif"></center>
<%@ include file="link.jsp"%>
<b>总邮件数为:<%=msgCount%></b>
<b>未读邮件数为:<%=unReadCount%></b>

<form ACTION="listone.jsp">
<table cellpadding=1 cellspacing=1 width="75%" border=1 align=center>
<tr bgcolor="ffffcc">
<td width="5%" ></td>
<td width="35%" align=center><b>发送者</b></td>
<td width="20%" align=center><b>日期</b></td>
<td width="30%" align=center><b>主题</b></td>
<td width="10%" align=center><b>大小</b></td>
</tr>

<%
  Message m = null;
  // for each message, show its headers
  for (int i = 1; i <= msgCount; i++) {
      m = f.getMessage(i);

      //  如果邮件是已被删除的，就不显示它了
      if (m.isSet(Flags.Flag.DELETED))
          continue;
%>

   <%--opt --%>
   <tr valign=middle >
   <td width=5% align=center><input TYPE=CHECKBOX NAME="delIndex<%=i%>"></td>
```

```
    <%-- from --%>
    <td width="35%" align=center>
    <%    if(!m.isSet(Flags.Flag.SEEN)) out.print("<b>"); %>
    <% out.println((m.getFrom() != null) ? m.getFrom()[0].toString() : " "); %>
    <% if(!m.isSet(Flags.Flag.SEEN))out.print("</b>"); %>
     </td>

     <%--date --%>
     <td width="20%" align=center>
     <%if(!m.isSet(Flags.Flag.SEEN))out.println("<b>");%>
     <%=m.getSentDate() %>
     <%if(!m.isSet(Flags.Flag.SEEN))out.println("</b>");%>
     </td>

     <%--subject & link --%>
     <td width="30%" align=center>
     <%
     String link=" ";
     if(f.getName().equals("Draft")){
       link="compose.jsp?edit=true";
       userInfo.setCurrentMsg(m);
     }
     else link="showmail.jsp" + "?messageindex=" + i;

     if(!m.isSet(Flags.Flag.SEEN))out.println("<b>");
       out.println("<a href="+link+">" +
        ((m.getSubject() != null)&& !m.getSubject().equals(" ") ?
                                   m.getSubject() : "<i>No Subject</i></a>"));
      if(!m.isSet(Flags.Flag.SEEN))out.println("</b>");
     %>
     </td>

   <%-- size--%>
   <td width="10%" align=center>
   <%
     if(!m.isSet(Flags.Flag.SEEN))out.println("<b>");
     out.println(m.getSize()+"Bytes");
     if(!m.isSet(Flags.Flag.SEEN))out.println("</b>");
    %>
    </td>
    </tr>
<%
 }
%>

<p><input TYPE="SUBMIT" NAME="submit" VALUE="删除选中邮件">
</table></form>
</body>
</html>
```

12.4.6　查看邮件

showmail.jsp 用于显示邮件的内容，它根据客户提供的请求参数决定显示哪封邮件，这个请求参数就是邮件在邮件夹中的序号，这个序号从 1 开始(不是从 0)。下面是 showmail.jsp 的源代码：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import ="cn.ac.ict.JavaMail.*,javax.mail.Store"%>
<%@ page import ="javax.mail.Folder"%>
<%@ page import ="javax.mail.URLName"%>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import ="cn.ac.ict.*,javax.mail.Store"%>
<%@ page import ="javax.mail.Folder"%>
<%@ page import ="javax.mail.URLName"%>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
  <head>
    <title>查看邮件</title>
  </head>
<body bgcolor="#FFFFFF">
<jsp:useBean id="userInfo" scope="session" class="cn.ac.ict.JavaMail.MailUserInfoBean" />

<%
          Folder folder= userInfo.getCurrentFolder();
          Message currmsg=null;
          int msgNum=1;
          //获取邮件索引
          String messageindex=request.getParameter("messageindex");
          //获取邮件
          if(messageindex!=null){
          msgNum=Integer.parseInt(messageindex);
          currmsg=folder.getMessage(msgNum);
          userInfo.setCurrentMsg(currmsg);
          }else
          currmsg = userInfo.getCurrentMsg();
          currmsg.setFlag(Flags.Flag.SEEN,true);
          if(currmsg==null){
%>
<jsp:forward page="error.jsp" />
<%}%>
<center>
<font size="+3"><b>
```

```
    <%    out.println(JMailUtil.getChinese(folder.getName())+"  邮件  "); %>
</b></font></center><p>

<%@ include file="link.jsp"%><p>
</center>
<center>
<a href="write.jsp?reply=true" >回复</a>
<a href="listone.jsp?delIndex<%=msgNum%>=on" >删除该邮件</a>

<%-- 下面开始显示邮件 --%>
    <table width=75% align=center border=1%>
    <tr><td>

<tr><td><b>发信人:</b></td><td> <%=JMailUtil.AddressToString(currmsg.getFrom())%><br></td></tr>
<tr><td><b>收信人:</b></td><td> <%=JMailUtil.AddressToString(currmsg.getRecipients(Message.
RecipientType.TO))%><br></td></tr>
<tr><td><b> 抄 送 :</b>   </td><td><%=JMailUtil.AddressToString(currmsg.getRecipients(Message.
RecipientType.CC))%><br></td></tr>
<tr><td><b>日期:</b> </td><td><%=currmsg.getSentDate()%><br></td></tr>
<tr><td><b>主题:</b></td><td> <%=currmsg.getSubject()%><br></td></tr>
<tr><td><b>邮件内容:</b></td><td>
<%
out.print(currmsg);
 %><br></td></tr>
</td></tr></table>
</body>
</html>
```

程序运行的结果如图 12.8 所示。



图 12.8　查看邮件的内容

用户可以通过上面的链接对邮件进行操作。单击【回复】链接会把页面转到 write.jsp；单击【删除该邮件】链接，会把控制权交给 listone.jsp 来处理，删除这个邮件。

12.4.7　写新邮件

write.jsp 是用于创建和发送邮件的页面。当用户单击常用链接的【写新邮件】链接时，

或当用户回复某个邮件时，都会把请求转发到这个页面。这个页面的效果如图 12.9 所示。

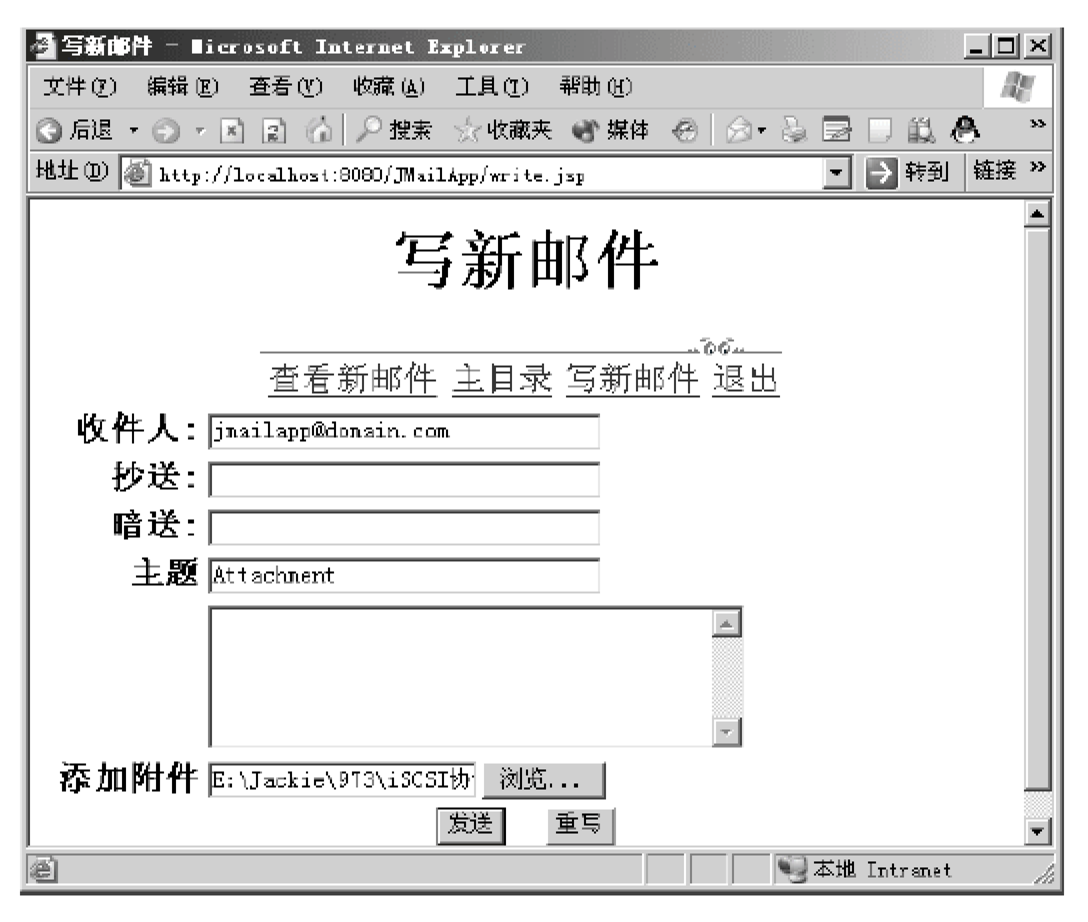


图 12.9　写新邮件

在这个页面写好邮件后，会提交给 write.jsp 页面，对其进行相应的操作。下面是 write.jsp 的源代码：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import ="cn.ac.ict.JavaMail.*,javax.mail.Store"%>
<%@ page import ="javax.mail.Folder"%>
<%@ page import ="javax.mail.URLName"%>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="javax.mail.Message.RecipientType" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
  <head>
    <title>写新邮件</title>
  </head>
<body bgcolor="#FFFFFF">
<jsp:useBean id="userInfo" scope="session" class="cn.ac.ict.JavaMail.MailUserInfoBean" />

<%
String operation=request.getParameter("operation");
//下面获取邮件的相关信息，如发件人和收件人等
String reply=request.getParameter("reply");
String edit=request.getParameter("edit");

String to = request.getParameter("to");
String cc = request.getParameter("cc");
String bcc = request.getParameter("bcc");
String subj = request.getParameter("subject");
String text = request.getParameter("text");
String attachment = request.getParameter("attachment");
```

```
if((operation!=null)&&(operation.equals("send"))){
        attachment= new String(attachment.getBytes("ISO8859-1"),"GB2312");
        Message msg = userInfo.createMessage(to,cc,bcc,subj,text,attachment);
        //查看邮件是否已经成功发送
        if(userInfo.sendMessage(msg)==JMailUtil.SUCCESS){
                out.println("The Mail has been sent to "+to+"!");
        }else{
                out.println("The Mail sent to "+to+" Failed!");
        }
}
String sto,scc,sbcc,subject,content;
sto = " ";
scc = " ";
sbcc = " ";
subject = " ";
content = " ";
%>
<p><center><font face="Arial,Helvetica" font size=+3><b>写新邮件</b></font></center></p>
<center><img src="images\signature.gif"></center>
<%@ include file="link.jsp"%>

<form action="write.jsp" method="get" enctype="multipart/form-data">
<table border="0" width="100%">
<tr>
        <td width="16%" height="22"><p align="right"><b>收件人:</b></td>
        <td width="84%"height="22"><input type="text"name="to"value="<%=sto%>"size="30" ></td>
</tr>
<tr>
        <td width="16%"><p align="right"><b>抄送:</b></td>
        <td width="84%"><input type="text" name="cc" value="<%=scc%>" size="30"></td>
</tr>
<tr>
        <td width="16%"><p align="right"><b>暗送:</b></td>
        <td width="84%"><input type="text" name="bcc" value="<%=sbcc%>" size="30"></td>
</tr>
<tr>
        <td width="16%"><p align="right"><b>主题</b></td>
        <td width="84%"><input type="text" name="subject" value="<%=subject%>" size="30"></td>
</tr>
<tr>
        <td width="16%"> </td>
        <td width="84%"><textarea name="text" rows="5" cols="40"><%=content%></textarea>
</td>
</tr>
<tr>
        <td width="16%"><p align="right"><b>添加附件</b></td>
        <td width="84%"> <input type="file" name="attachment"></td></tr>
```

```
</table>
<center>
<b>
        <input type="hidden" name="operation" value="send">
        <input type="submit" name="submit" value="发送"> 
        <input type="reset" name="reset" value="重写">
</center>
</form>
</body>
</html>
```

按照如图 12.9 的输入提交后，一封邮件就被递送到 jmailapp@domain.com。如果使用 Foxmail 查看，效果如图 12.10 所示。

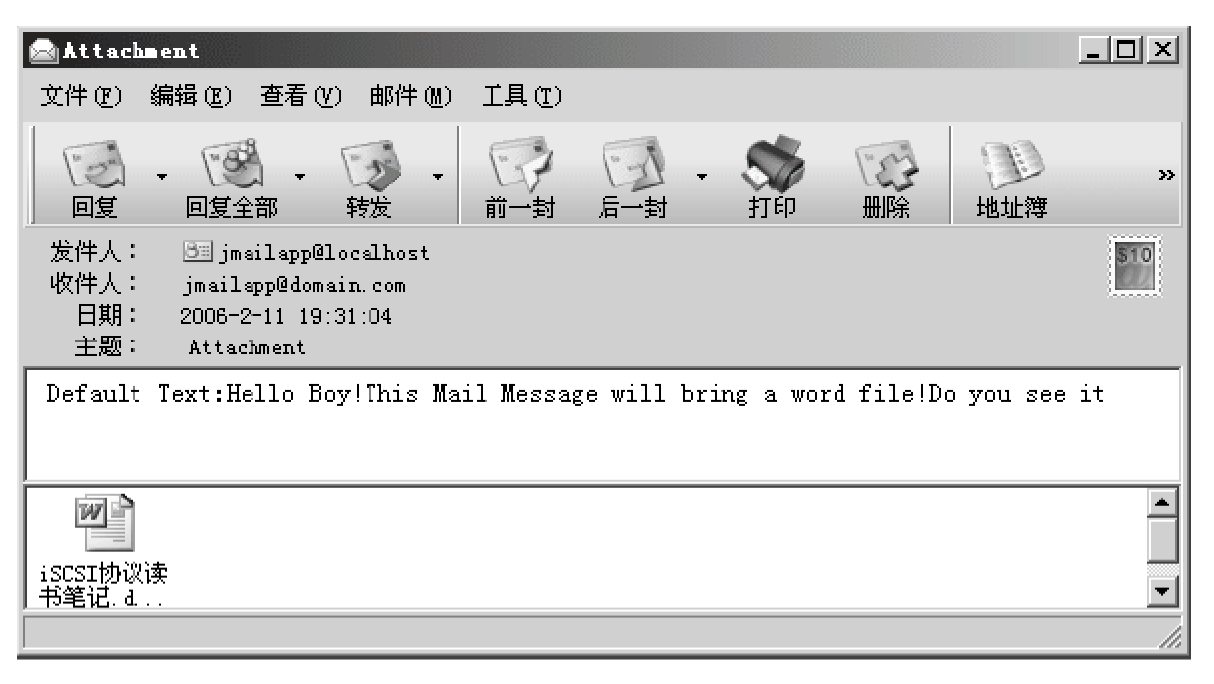


图 12.10　发送邮件

12.4.8　退出系统

logout.jsp 是系统的退出页面，结束会话。断开客户和邮件系统的链接是通过以下语句实现的：

```
String username=userInfo.getURLName().getUsername();
userInfo.getStore().close();
session.invalidate();
```

12.4.9　发布 Java Mail Web 应用

按照上面的介绍，把需要编写的文件编写好后，应执行以下操作：

（1）将 mail.jar、activation.jar 和 log4j-1.2.11.jar 文件放在本应用的 WEB-INF/lib 目录下。

（2）将编译使用到的 Java 文件放在 WEB-INF/classes 目录下。

把各种文件放在合适的位置后，这个应用配置好的整个文件夹的结构如图 12.11 所示。

最后把整个 JMailApp 文件夹复制到<TOMCAT_HOME>\webapps 目录下，然后重新启动 Tomcat，在浏览器地址栏中输入地址 http://localhost:8080/JMailApp/login.jsp，可以看到页面显示如图 12.6 所示。

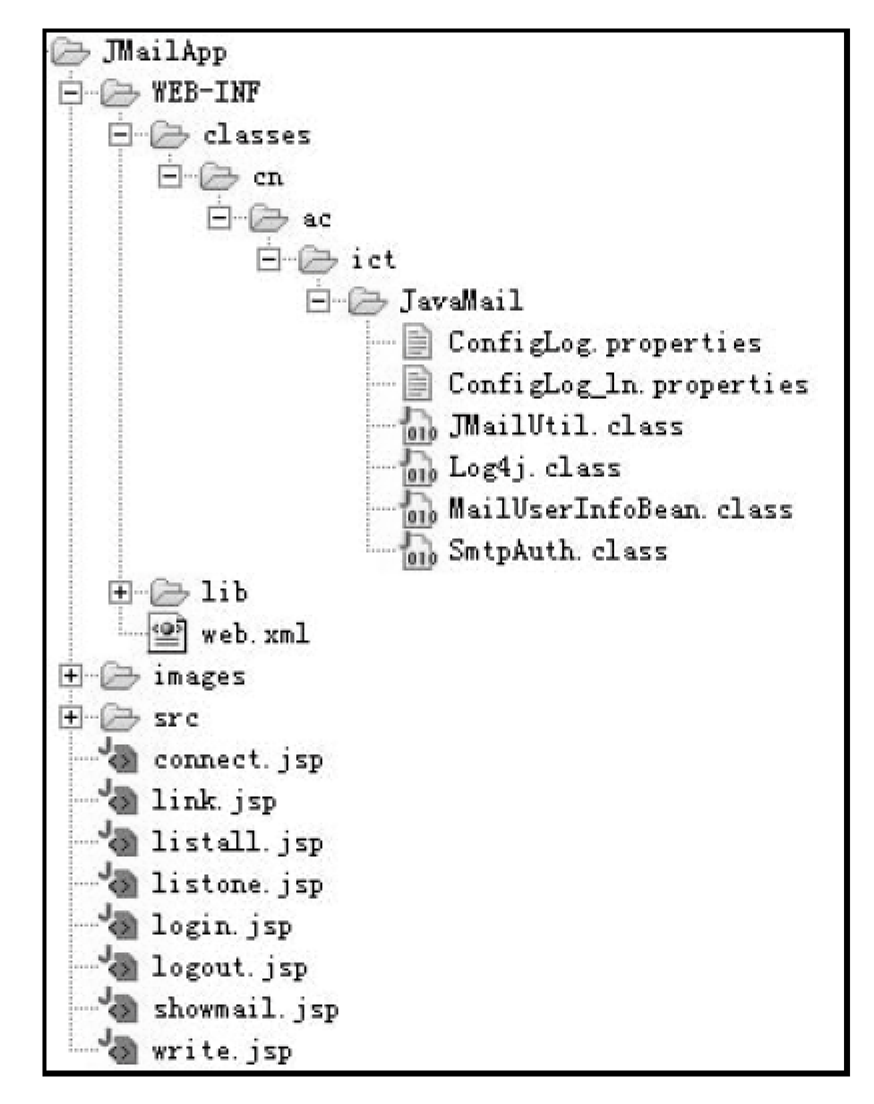


图 12.11　Java Mail Web 应用程序结构

12.5　小　　结

Java Mail API 是 Sun 开发的最新标准扩展 API 之一，它给 Java 应用程序开发者提供了独立于平台和协议的邮件/通信解决方案。Java Mail API 实际上依赖于另外一个 Java 扩展 JAF，即 JavaBeans 活动框架（JavaBeans Activation Framework）。JAF 的目的在于统一处理不同数据格式的方法（不论数据格式为简单文本还是由图片、声音、视频甚至其他“活动”内容共同组成的复合文档）。在这个意义上，JAF 对 Java 的作用正如插件对 Web 浏览器的作用。

本章介绍了 Java Mail API 的应用以及邮件协议的相关知识，通过本章的学习，读者对 Java Mail API 应该有比较好的理解。本章还使用一个功能比较齐全的例子讲述了 Java Mail API 的应用，在这个例子中，客户可以使用这个应用收发邮件以及对自己的邮箱进行管理等各种操作。

本章介绍了如何发送文本格式的邮件、带附件的文本格式的邮件。读者可以尝试发送 HTML 格式的邮件，这样多多练习才能更好地掌握 Java Mail 编程的技巧。

第 13 章　XML 在 JSP 中的应用

随着 XML 语言的广泛应用，越来越多的应用中需要使用 XML 进行数据的交换，在使用 JSP 技术开发过程中同样不可避免地要处理大量的 XML 文档，JSP 技术和 XML 技术成为开发 Web 应用的两个很重要的工具，本章将介绍在使用 JSP 和 XML 技术进行开发过程中需要使用的技术。

13.1　XML 与 JSP

13.1.1　什么是 XML

XML 即可扩展标记语言（eXtensible Markup Language）。标记是指计算机所能理解的信息符号，通过此种标记，计算机之间可以处理包含各种信息的文章等。

XML 在写法上很类似于 HTML，它属于 SGML 的子集，继承 SGML 自定义标记的优点，并且删除一些 SGML 复杂的部分，在功能上能够弥补 HTML 标记的不足，拥有更多的扩展性。

不过 XML 并不是用来编排内容的，而是用来描述数据的，它并没有如同 HTML 一般的默认标记，用户需要自己定义描述数据所需的各种标记。

注意：（1）XML 并不是 HTML 的替代产品，也不是 HTML 的升级，它只是 HTML 的补充，为 HTML 扩展更多功能。HTML 仍将在较长的一段时间内继续被使用，但 HTML 的升级版本 XHTML 的确正在向适应 XML 靠拢。

（2）不能用 XML 来直接写网页。即便是包含了 XML 数据，依然要转换成 HTML 格式才能在浏览器上显示。

下面是一个很简单的 XML 文件：

```
<?xml version="1.0" encoding="GB2312"?>
<user>
    <firstname>Jackie</firstname >
    <lastname>pter</lastname >
    <email>jmailappuser@163.com </email>
    <registerdate>20060115</ registerdate >
</ user >
```

第一行指明这是一个 XML 文件，版本为 1.0，使用的编码类型为简体中文 GB2312，第二行是根节点，根节点下拥有多个子节点，表示一个用户的信息。

读者可以把这个文件与下面的 HTML 文件相比较：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
    <title>用户信息</title>
  </head>

<body>
  <h1>用户信息</h1>
  <h3>firstname：Jackie</h3>
  <h3>lastname：pter</h3>
  <h3>email：jmailappuser@163.com</h3>
  <h3>registerdate：20060115</h3>
</body>
</html>
```

XML 文件除了没有使用 HTML 标记以外，在结构和内容上都非常相似，虽然两者表示的内容是一样的，都是用户的信息，但当在浏览器中浏览这两个页面时，可以看到它们是有区别的，HTML 的页面显示如图 13.1 所示，XML 文件的页面显示如图 13.2 所示。



图 13.1　HTML 文件页面

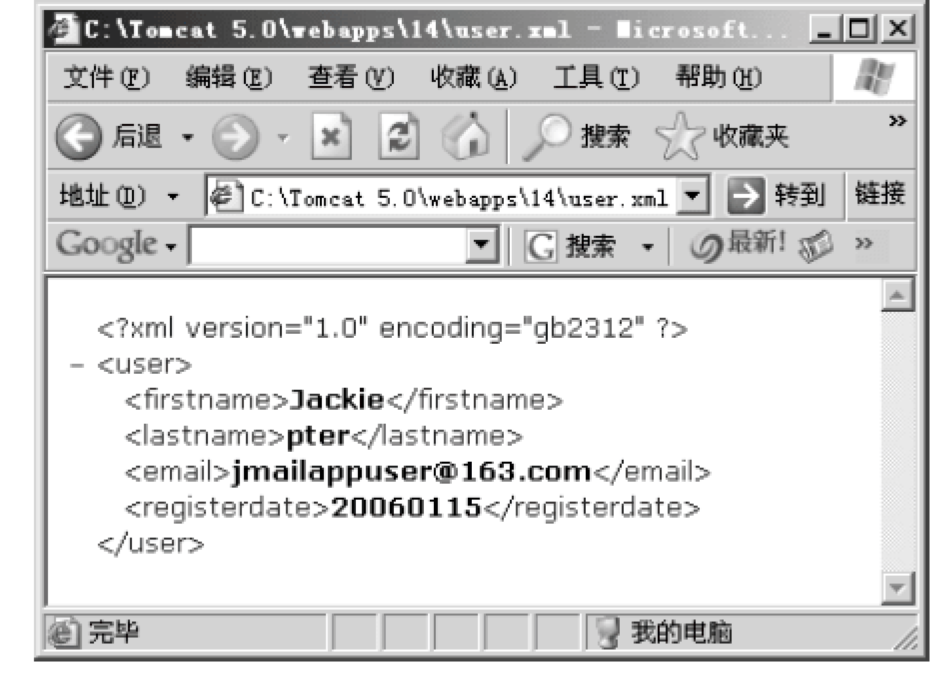


图 13.2　XML 文件页面

在 HTML 文件页面显示中可以很明确地知道这里要描述一个用户的信息，但看 XML 文件的页面时却要费好大劲才能理解页面表示的是什么内容，因为 XML 文件并不负责如何表示数据，它只是负责描述数据。

13.1.2　XML 的特点

从 13.1.1 节中可以了解到 XML 文件的基本架构是很简单的，除了标记名称需要用户自己定义外，写法跟 HTML 文件没有什么区别。XML 文件有几个特点需要注意，下面分别介绍 XML 文件的特点。

1．XML 文件是格式良好的

XML 文件是格式良好的，与 HTML 文件相比，XML 文件的标记都必须要有一个结束标记，例如：

```
<email>jmailappuser@163.com </email>
```

该代码的开头有一个<email>标记，末尾就必须要有一个</email>的结束标记，简单地说，标记必须是成双成对的，如果 XML 标记没有内容，只有属性，XML 标记的写法与 HTML 的稍有不同，结束的“>”符号前要有“/”符号，如下：

```
<needconfirm value ="true"/>
```

2．XML 文件需要验证

因为 XML 文件的标记是由用户自行定义的，XML 文件并没有任何默认标记和架构，只是在开头声明这是一个 XML 文件，所以要使用 DTD（Document Type Defination）或 XMLSchema 检查 XML 标记的定义是否符合语法。

XML 提供文件验证的机制，其目的是检查 XML 文件是否符合自行定义的标记规则，因为 XML 的标记并没有如同 HTML 那样，已经替标记预先定义用途。例如：看到 HTML 的<P>标记就知道内含的文字是一个段落，<H>标记是标题文字，至于 XML 的标记如果没有验证机制，那么文件是否正确根本无从得知。

如果 XML 文件主要是提供网站内存或数据的交换，就可以通过自行定义的验证机制，任何人只需依照规则编写 XML 文件，都可以使用相同的机制检查 XML 文件是否符合规则，只需通过验证就可以提供网站数据或符合交换数据的标准格式。

13.1.3　XML 与 JSP 的工具介绍

在 Java 语言中使用对象表示数据，XML 是一个标记语言，但它本身什么都不做，因此 Java 要使用其中的数据，必须要先解析 XML 文件。随着 XML 和 Java 的流行，现在已经有很多工具可以用于解析和处理 XML 文件。

下面简单介绍几种常用的工具，有的在本书中的例子中也会用到，这些工具大部分都是开放源代码的。

1．XSLT

XSLT 是扩展样式表转换语言（Extensible Stylesheet Language Transformations）的简称，这是一种对 XML 文档进行转化的语言，XSLT 语言本身也是什么都不做的，它对 XML 文件的转换还需要依赖其他软件工具的帮助。

根据 W3C 的规范说明书（http://www.w3.org/TR/xslt），最早设计 XSLT 的用意是帮助 XML 文档（document）转换为其他文档。但随着技术的进步，XSLT 已不仅仅用于将 XML 转换为 HTML 或其他文本格式，更全面的定义应该是：XSLT 是一种用来转换 XML 文档结构的语言。

2．JAXP

JAXP 是 Java 语言中用于简化 XML 处理的 API，它是 Java API for XML Processing 的英文字头缩写，JAXP 并不是一个 XML 文件的解析器，JAXP 支持 DOM、SAX、XSLT 等标准。为了增强 JAXP 使用上的灵活性，开发者特别为 JAXP 设计了一个 Pluggability Layer，

在 Pluggability Layer 的支持下，JAXP 既可以和具体实现 DOM API、SAX API 的各种 XML 解析器（XML Parser，例如 Apache Xerces）联合工作，又可以和具体执行 XSLT 标准的 XSLT 处理器（XSLT Processor，例如 Apache Xalan）联合工作。

读者如果需要深入学习或下载相关文档，可以访问 Sun 的网站：http://java.sun.com/xml/download.html。

3. DOM

DOM 是 Document Object Model 的缩写，即文档对象模型。前面说过，XML 将数据组织为一棵树，所以 DOM 就是对这棵树的一个对象描述。也就是通过解析 XML 文档，为 XML 文档在逻辑上建立一个树模型，树的节点是一个个对象。程序员通过存取这些对象就能够存取 XML 文档的内容。

4. JDOM

JDOM 就是 Java 与 DOM 的结合体，它通过只实现 DOM 中最重要和最普遍的部分简化了 DOM，使得 JDOM 使用起来更加简单和快速，虽然它不具有 DOM 的所有特点，但对一个 Java-XML 的开发者而言已经足够了，读者如果需要深入学习或下载相关文档，可以访问网站：http://www.jdom.org/。

5. SAX

SAX 是 Simple API for XML 的缩写，它并不是由 W3C 官方所提出的标准，可以说是“民间”的事实标准。实际上，它是一种社区性质的讨论产物。不同于 DOM 的文档驱动，它是事件驱动的。它并不需要读入整个文档，而文档的读入过程也就是 SAX 的解析过程。

6. dom4j

dom4j 是一个非常优秀的 Java XML API，具有性能优异、功能强大和极端易于使用的特点，同时它也是一个开放源代码的软件。如今越来越多的 Java 软件都在使用 dom4j 来读写 XML，Sun 的 JAXM 也在用 dom4j。读者可以到如下网站下载 dom4j：http://dom4j.sourceforge.net。

13.2　使用 DOM 解析接口操作 XML 文件

在上一节中对 DOM 作了一个简单的介绍，在这一节就通过一个实际的例子和 DOM API 的相关知识演示如何使用 DOM 解析接口操作 XML 文件。

13.2.1　DOM API

DOM 的基本对象有 5 个：Document、Node、NodeList、Element 和 Attr。下面就这些对象的功能和实现的方法作一个大致的介绍。

1．Document 对象

Document 对象代表了整个 XML 的文档，所有其他的 Node 都以一定的顺序包含在 Document 对象之内，排列成一个树型的结构，程序员可以通过遍历这棵树来得到 XML 文档的所有内容，这也是对 XML 文档操作的起点。在实际开发时总是先通过解析 XML 源文件而得到一个 Document 对象，然后再来执行后续的操作。此外，Document 还包含了创建其他节点的方法，比如 createAttribut()用来创建一个 Attr 对象。

它所包含的主要方法有：

- createAttribute(String)：用给定的属性名创建一个 Attr 对象，并可在其后使用 setAttributeNode 方法来放置在某一个 Element 对象上面。
- createElement(String)：用给定的标签名创建一个 Element 对象，代表 XML 文档中的一个标签，然后就可以在这个 Element 对象上添加属性或进行其他的操作。
- createTextNode(String)：用给定的字符串创建一个 Text 对象，Text 对象代表了标签或者属性中所包含的纯文本字符串。如果在一个标签内没有其他的标签，那么标签内的文本所代表的 Text 对象是这个 Element 对象的惟一子对象。
- getElementsByTagName(String)：返回一个 NodeList 对象，它包含了所有给定标签名字的标签。
- getDocumentElement()：返回一个代表这个 DOM 树的根节点的 Element 对象，也就是代表 XML 文档根元素的那个对象。

2．Node 对象

Node 对象是 DOM 结构中最为基本的对象，代表了文档树中的一个抽象的节点。在实际使用时，很少会真正地用到 Node 这个对象，而是用到诸如 Element、Attr、Text 等 Node 对象的子对象来操作文档。Node 对象为这些对象提供了一个抽象的、公共的根。虽然在 Node 对象中定义了对其子节点进行存取的方法，但有一些 Node 子对象，比如 Text 对象，它并不存在子节点，这一点是要注意的。Node 对象所包含的主要方法有：

- appendChild(org.w3c.dom.Node)：为这个节点添加一个子节点，并放在所有子节点的最后，如果这个子节点已经存在，则先把它删掉再添加进去。
- getFirstChild()：如果节点存在子节点，则返回第一个子节点，相应地，还有 getLastChild()方法返回最后一个子节点。
- getNextSibling()：返回在 DOM 树中这个节点的下一个兄弟节点，相应地，还有 getPreviousSibling()方法返回其前一个兄弟节点。
- getNodeName()：根据节点的类型返回节点的名称。
- getNodeType()：返回节点的类型。
- getNodeValue()：返回节点的值。
- hasChildNodes()：判断是否存在子级节点。
- hasAttributes()：判断这个节点是否具有带有指定名称的属性。
- getOwnerDocument()：返回节点所处的 Document 对象。
- insertBefore(org.w3c.dom.Node new，org.w3c.dom.Node ref)：在给定的一个子对象

前再插入一个子对象。

- ❑ removeChild(org.w3c.dom.Node)：删除给定的子节点对象。
- ❑ replaceChild(org.w3c.dom.Node new，org.w3c.dom.Node old)：用一个新的 Node 对象代替给定的子节点对象。

3．NodeList 对象

NodeList 对象，是代表一个包含了一个或者多个 Node 对象的列表，可以简单地把它看成一个 Node 的数组。可以通过下列两种方法来获得列表中的元素：

- ❑ GetLength()：返回列表的长度。
- ❑ Item(int)：返回指定位置的 Node 对象。

4．Element 对象

Element 对象代表的是 XML 文档中的标签元素，继承于 Node，亦是 Node 的最主要的子对象。在标签中可以包含有属性，因而 Element 对象中有存取其属性的方法，而任何 Node 中定义的方法，也可以用在 Element 对象上面。

- ❑ getElementsByTagName(String)：返回一个 NodeList 对象，它包含了在这个标签中其下的子孙节点中具有给定标签名字的标签。
- ❑ getTagName()：返回一个代表这个标签名字的字符串。
- ❑ getAttribute(String)：返回标签中给定属性名称的属性的值。需要注意的是，因为 XML 文档中允许有实体属性出现，而这个方法对这些实体属性并不适用。这时需要用到 getAttributeNodes()方法得到一个 Attr 对象来进行进一步的操作。
- ❑ getAttributeNode(String)：返回一个代表给定属性名称的 Attr 对象。

5．Attr 对象

Attr 对象代表某个标签中的属性。Attr 继承于 Node，但因为 Attr 实际上是包含在 Element 中的，它并不能被看作是 Element 的子对象，因而在 DOM 中 Attr 并不是 DOM 树的一部分，所以 Node 中的 getparentNode()、getpreviousSibling()和 getnextSibling()返回的都将是 null。也就是说，Attr 其实是被看作包含它的 Element 对象的一部分，它并不作为 DOM 树中单独的一个节点出现。这一点在使用时要同其他的 Node 子对象相区别。

上面所说的 DOM 对象在 DOM 中都是用接口定义的，在定义时使用的是与具体语言无关的 IDL 语言来定义的。因而，DOM 其实可以在任何面向对象的语言中实现，只要它实现了 DOM 所定义的接口和功能就可以了。同时，有些方法在 DOM 中并没有定义，是用 IDL 的属性来表达的，当被映射到具体的语言时，这些属性被映射为相应的方法。

13.2.2　使用 DOM 读写 XML 文件的例子

下面介绍一个使用 DOM 读写 XML 文件的例子，读者可以按照下面介绍的步骤开发这个例子。

1．编写 XML 文件

下面编写一个 books.xml 文件，在这个文件中有很多本书的信息，在后面介绍 JDOM 和

SAX 技术时也使用这个 XML 文件。books.xml 文件的内容如下：

```
<?xml version="1.0" encoding="gb2312"?>
<books>
	<book>
		<title>JSP 应用开发指南</title>
		<url newWindow="no">http://java.sun.com</url>
		<author>Author 001</author>
		<date>
			<day>23</day>
			<month>1</month>
			<year>2006</year>
		</date>
		<description>一本详尽介绍 JSP 技术的书</description>
	</book>

	<book>
		<title>Ant 技术详解</title>
		<url newWindow="no">http://ant.apache.org/</url>
		<author>Author 002</author>
			<date>
				<day>23</day>
				<month>1</month>
				<year>2006</year>
			</date>
		<description>一本详尽介绍 Ant 技术的书</description>
	</book>

	<book>
		<title>Apache 技术详解</title>
		<url newWindow="no">http://www.apache.org/</url>
		<author>Author 003</author>
		<date>
			<day>23</day>
			<month>1</month>
			<year>2006</year>
		</date>
		<description>一本详尽介绍 Apache 技术的书</description>
	</book>

</books>
```

2．编写读写 XML 文档的 JSP 页面

准备好要读写的 XML 文件后，就可以开始编写读写 XML 文档的 JSP 页面了，下面是读写 XML 文件的 JSP 页面（DOMExample.jsp），代码如下：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import="javax.xml.parsers.*"%>
<%@ page import="org.w3c.dom.*"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
```

```
<html>
  <head>
    <title>DOM 操作 XML 文件  JSP</title>
  </head>
<body bgcolor="#FFFFFF">
<%
//建立一个解析器工厂
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
//获得一个具体的解析器对象
      DocumentBuilder builder=factory.newDocumentBuilder();
//对 XML 文档进行解析，获得 Document 对象
      Document doc=builder.parse("C:/Tomcat 5.0/webapps/13/books.xml");
      doc.normalize();
//获取所有的 book 元素列表
      NodeList books = doc.getElementsByTagName("book");
      %>
<h2>图书列表</h2><br>
<%
      for (int i=0;i<books.getLength();i++){
//获取一个 book 元素
            Element book=(Element) books.item(i);
//以下获取 book 的子元素，并输出
            out.print("title: ");

out.println(book.getElementsByTagName("title").item(0).getFirstChild().getNodeValue());
out.print("<br>");
out.print("URL: ");
out.println(book.getElementsByTagName("url").item(0).getFirstChild().getNodeValue());
out.print("<br>");
out.print("Author: ");

out.println(book.getElementsByTagName("author").item(0).getFirstChild().getNodeValue());
out.print("<br>");
out.print("Date: ");
Element bookdate=(Element) book.getElementsByTagName("date").item(0);
String day=bookdate.getElementsByTagName("day").item(0).getFirstChild().getNodeValue();
String month=bookdate.getElementsByTagName("month").item(0).getFirstChild().getNodeValue();
String year=bookdate.getElementsByTagName("year").item(0).getFirstChild().getNodeValue();
out.println(day+"-"+month+"-"+year);
out.print("<br>");
out.print("Description: ");

out.println(book.getElementsByTagName("description").item(0).getFirstChild().getNodeValue());
out.print("<br>");
out.print("<br>");
      }
%>
</body>
</html>
```

要读取 XML 文件的内容，首先要把 XML 文件的内容解析为一个个的对象供程序使用，这需要建立一个解析器工厂，以利用这个工厂来获得一个具体的解析器对象：

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

使用 DocumentBuilderFactory 的目的是为了创建与具体解析器无关的程序，当 DocumentBuilderFactory 类的静态方法 newInstance()被调用时，它根据一个系统变量来决定具体使用哪一个解析器。又因为所有的解析器都服从于 JAXP 所定义的接口，所以无论具体使用哪一个解析器，代码都是一样的。因此当在不同的解析器之间进行切换时，只需要更改系统变量的值，而不用更改任何代码。这就是工厂所带来的好处。

```
DocumentBuilder builder=factory.newDocumentBuilder();
```

当获得一个工厂对象后，使用它的静态方法 newDocumentBuilder()可以获得一个 DocumentBuilder 对象，这个对象代表了具体的 DOM 解析器。但具体是哪一种解析器对于程序而言并不重要。

然后，就可以利用这个解析器来对 XML 文档进行解析了：

```
Document doc=builder.parse("C:/Tomcat 5.0/webapps/13/books.xml");
doc.normalize();
```

DocumentBuilder 的 parse()方法接受一个 XML 文档名作为输入参数，返回一个 Document 对象，这个 Document 对象就代表了一个 XML 文档的树模型。以后所有对 XML 文档的操作都与解析器无关，直接在这个 Document 对象上进行操作就可以了。而具体对 Document 操作的方法是由 DOM 所定义的，这在 13.2.1 节中已经介绍了。

注意： 对 Document 对象调用 normalize()，可以去掉 XML 文档中作为格式化内容的空白而映射在 DOM 树中的不必要的 Text Node 对象。否则得到的 DOM 树可能并不如所想象的那样。特别是在输出时，这个 normalize()更为有用。

XML 文档中的空白符也会被作为对象映射在 DOM 树中。因而，直接调用 Node 方法的 getChildNodes 方法有时会有些问题，有时不能够返回所期望的 NodeList 对象。解决的办法是使用 Element 的 getElementByTagName(String)，返回的 NodeList 就是所期待的对象了。然后可以用 item()方法提取想要的元素。

```
	NodeList books = doc.getElementsByTagName("book");
	%>
<h2>图书列表</h2><br>
<%
	for (int i=0;i<books.getLength();i++){
		Element book=(Element) books.item(i);
……
```

把 books.xml 文件和 DOMExample.jsp 文件复制到本章的 Web 应用目录下，然后在浏览器地址栏中输入如下地址：http://localhost:8080/13/DOMExample.jsp，可以看到页面显示如图 13.3 所示。

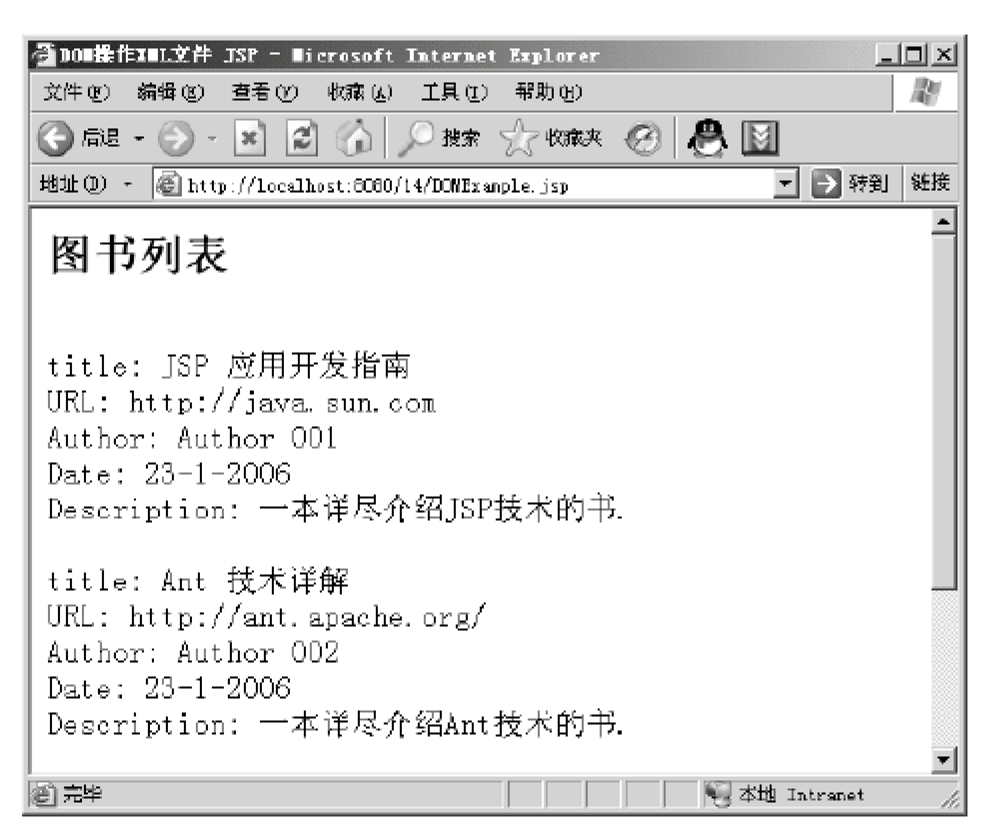


图 13.3　DOM 读取 XML 文件

13.3　使用 JDOM 操作 XML 文件

13.3.1　JDOM 的安装与简介

1．JDOM 的安装

由于 JDOM 不是 J2SE 的一部分，因此如果要使用 JDOM 读写 XML 文件，需要到 http://jdom.org 下载 JDOM 的类库。

目前 JDOM 的最新版本是 1.0，将下载的压缩包解压缩后，把解压 build 目录下的 jdom.jar 和 lib 目录下的 ant.jar、jaxen-core.jar、jaxen-jdom.jar、saxpath.jar、xalan.jar、xerces.jar、xml-apis.jar 文件复制到 Web 应用的 WEB-INF\lib 目录下。

2．JDOM 简介

JDOM 的处理方式有些类似于 DOM，但它主要是用 SAX 实现的，不需要担心处理速度和内存的问题。另外，JDOM 中的接口很少，全部是类，也没有类工厂类。其最重要的一个包 org.jdom 中主要有以下类：

- Attribute
- CDATA
- Comment
- DocType
- Document
- Element
- EntityRef
- Namespace
- ProcessingInstruction
- Text

这些对象和 XML 文件中的元素是一一对应的，这里就不详细介绍了，读者可以在仔细

了解 XML 文件的基础上熟悉这些类的使用。

注意： 数据输入要用到 XML 文档，需要 org.jdom.input 包，反过来需要 org.jdom.output 包。

13.3.2　使用 JDOM 读写 XML 文件

1．编写 XML 文件

在本例中使用 13.2.2 节中编写的 XML 文件 books.xml，这里就不作介绍了。

2．编写读写 XML 文档的 JSP 页面

准备好要读写的 XML 文件后，就可以开始编写读写 XML 文档的 JSP 页面了，下面是读写 XML 文件的 JSP 页面（JDOMExample.jsp），代码如下：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import="java.io.*"%>
<%@ page import="java.util.*"%>
<%@ page import="org.jdom.*"%>
<%@ page import="org.jdom.output.*"%>
<%@ page import="org.jdom.input.*"%>
<%@ page import="javax.servlet.http.*"%>

<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
  <head>
    <title>DOM 操作 XML 文件  JSP</title>
  </head>
<body bgcolor="#FFFFFF">
<%
      Vector xmlVector = null;

      FileInputStream fi = null;
      try{
//建立文件输入流
            fi = new FileInputStream("C:/Tomcat 5.0/webapps/13/books.xml");
            xmlVector = new Vector();
//得到一个解析器对象
            SAXBuilder sb = new SAXBuilder();
//解析 XML 文件，获取一个 Document 对象
            Document doc = sb.build(fi);
//得到根元素
            Element root = doc.getRootElement();
//得到根元素所有子元素的集合
            List books = root.getChildren();
            Element book =null;
            out.print("<h2>图书列表</h2><br>");
            for(int i=0;i<books.size();i++){
//得到一本书元素
            book = (Element)books.get(i);
```

```
//获取子元素，并输出。
            out.print("title: ");
            out.print(book.getChild("title").getText());
            out.print("<br>");
            out.print("URL: ");
            out.print(book.getChild("url").getText());
            out.print("<br>");
            out.print("Author: ");
            out.print(book.getChild("author").getText());
            out.print("<br>");
            out.print("Date: ");
            Element bookdate=book.getChild("date");
            String day=bookdate.getChild("day").getText();
            String month=bookdate.getChild("month").getText();
            String year=bookdate.getChild("year").getText();
            out.println(day+"-"+month+"-"+year);
            out.print("<br>");
            out.print("Description: ");
            out.print(book.getChild("description").getText());
            out.print("<br>");
            out.print("<br>");
}

            }catch(Exception ex){

            }
%>

</body>
</html>
```

要读取 XML 文件的内容，首先要把 XML 文件的内容解析为一个个的对象供程序使用。在使用 DOM 读写 XML 文件时，首先建立一个工厂对象，通过工厂对象获取一个解析器，然后把 XML 文件解析为一个 Document 对象。但在使用 JDOM 时就不是这样了，它是把 XML 文件作为输入流，然后建立一个 SAX 的解析器，将其解析为一个 Document 对象，代码如下：

```
            fi = new FileInputStream("C:/Tomcat 5.0/webapps/13/books.xml");
            xmlVector = new Vector();
            SAXBuilder sb = new SAXBuilder();
            Document doc = sb.build(fi);
```

得到 XML 文件的 Document 对象后就可以得到文件的根元素和根元素下的所有子元素，然后对这些子元素进行循环解析就可以了。

```
            Element root = doc.getRootElement(); //得到根元素
            List books = root.getChildren(); //得到根元素所有子元素的集合
            Element book =null;
            out.print("<h2>图书列表</h2><br>");
            for(int i=0;i<books.size();i++){
```

13.4　使用 SAX 操作 XML 文件

在本节中介绍使用 SAX 操作 XML 文件，由于 SAX 减少了对内存的需求，因而成为很多技术行家推崇使用的技术。

13.4.1　SAX API

1．SAX 事件

以下 SAX 事件是被经常用到的，它们都在 org.xml.sax 包的 HandlerBase 类中被定义。

- startDocument：表示文档开始。
- endDocument：表示文档结束。
- startElement：表示元素开始。当一对标记中的起始标记中的所有内容被处理后，解析器激发此事件。包括了标记名和其属性。
- endElement：表示元素结束。
- characters：包含字符数据，类似于 DOM 的一个 Text 节点。

还有一些其他的 SAX 事件：

- ignorableWhitespace：此事件类似于前面所讨论的无用 DOM 节点。它与 character 事件相比，好处是：如果不需要空格符，开发者可以通过忽略这个事件来忽略所有的空格符。
- warning、error 和 fatalError 这 3 个事件表示了解析错误。开发者可根据需要来响应它们。
- setDocumentLocator：这个事件允许存储一个 SAX 的 Locator 对象。Locator 对象可以用来找出在文档中确切发生事件的地方。

2．SAX 处理的工作过程

SAX 是基于事件的一种处理 XML 文件的机制，下面简单介绍 SAX 处理的工作过程。

SAX 分析经过其 XML 流，这非常像老式的自动收报机纸条。考虑以下 XML 代码片断：

```
<?xml version="1.0"?>
<samples>
<server>UNIX</server>
<monitor>color</monitor>
</samples>
```

SAX 处理器分析这段代码将生成以下事件：

- Start document
- Start element (samples)
- Characters (white space)
- Start element (server)

- ❑ Characters (UNIX)
- ❑ End element (server)
- ❑ Characters (white space)
- ❑ Start element (monitor)
- ❑ Characters (color)
- ❑ End element (monitor)
- ❑ Characters (white space)
- ❑ End element (samples)

SAX API 允许开发者捕获这些事件，并对它们进行操作。SAX 处理涉及以下几步：

（1）创建事件处理程序。

（2）创建 SAX 解析器。

（3）将事件处理程序分配给解析器。

（4）对文档进行解析，将每个事件发送给处理程序。

13.4.2　使用 SAX 读写 XML 文件

下面介绍一个使用 SAX 读 XML 文件的例子，在这个例子中解析一个 XML 文件，并把文件的内容输出到 JSP 页面上。

1．编写 XML 文件

在本例中使用 13.2 节中编写的 XML 文件 books.xml，这里就不作介绍了。

下面编写一个 Book Bean，它可以用来存取一个 book 对象的信息，并提供相应的 get 和 set 方法，由于它的属性都是 Simple 的，所以这里也不作详细介绍，读者有疑问可以参考本书“JavaBeans 在 JSP 中的应用”一章中的介绍。下面是 BookBean.java 的源代码：

```
package cn.ac.ict;

public class BookBean {
	private String title = " ";
	private String author = " ";
	private String url = " ";
	private String day = " ";
	private String month = " ";
	private String year = " ";
	private String description	= " ";

	public void setTitle(String newtitle){
		this.title = newtitle;
	}
	public void setAuthor(String newauthor){
		this.author = newauthor;
	}
	public void setUrl(String newurl){
```

```
            this.url = newurl;
        }
        public void setDay(String newday){
            this.day = newday;
        }
        public void setMonth(String newmonth){
            this.month = newmonth;
        }
        public void setYear(String newyear){
            this.year = newyear;
        }
        public void setDescription(String descr){
            this.description = descr;
        }
        public String getTitle(){
            return this.title;
        }
        public String getAuthor(){
            return this.author;
        }
        public String getUrl(){
            return this.url;
        }
        public String getDescription(){
            return this.description;
        }
        public String getDate(){
            return this.day+"-"+this.month+"-"+this.year;
        }
}
```

2．SAXHandlerBean.java 文件

SAXHandlerBean 是对 XML 文件进行解析的一个类，这个类扩展了 ContentHandler 接口，SAXHandlerBean 类必须实现 ContentHandler 接口中的方法。

下面是 SAXHandlerBean 类的源代码：

```
package cn.ac.ict;

import java.io.IOException;
import java.util.Enumeration;
import java.util.Hashtable;

import javax.servlet.http.HttpServletRequest;

import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;

public class SAXHandlerBean implements ContentHandler {
```

```
	private Hashtable table = new Hashtable();
	private BookBean book = null;
	private Locator locator;
	private String currentvalue;

	public SAXHandlerBean() {
		super();
	}
//解析 XML 文件
	public void parseXML(String uri) {
		try{
			System.out.println("正在分析中的 XML 文件: " + uri + "\n");
			XMLReaderparser =XMLReaderFactory.createXMLReader("org.apache.xerces.
parsers.SAXParser");
			//XMLReader parser = new SAXParser();
			//注册自己设计的内容处理器
			parser.setContentHandler(this);
			//注册错误处理器
			parser.setErrorHandler(new SAXErrorHandler());
			//分析文件
			parser.parse(uri);
			} catch(IOException ioe){
				System.out.println("文件读取错误："+ioe.getMessage());
				} catch(SAXException saxe){
					System.out.println("XML 分析错误：  "+saxe.getMessage());
					}
				}
	public static void main(String[] args) {
		Hashtable table = new Hashtable();
	   //如果参数数目不对，则输出使用说明，并结束程序
	   if( args.length != 1 ){
	      System.out.println("请输入欲分析的文件名: " +"java MySAXParser [XML URI]");
		 System.exit(-1);
	   }
	   String uri = args[0];
	   SAXHandlerBean myParser = new SAXHandlerBean();
	   myParser.parseXML(uri);
	   table = myParser.getTable();
	   Enumeration enum = table.keys();
	   while(enum.hasMoreElements()){
	     String booktitle = (String)enum.nextElement();
	     BookBean book = (BookBean)table.get(booktitle);
	     System.out.print(book.getDescription());
	   }
	  }

	public void setTable(Hashtable table){
		this.table = table;
	}
```

```
    public void getTable(HttpServletRequest request){
        request.getSession().setAttribute("books",table);
    }
    public Hashtable getTable(){
        return this.table;
    }
   public void setDocumentLocator (Locator locator){
        System.out.println("设置 Locator 对象…");
        //把 SAX 的 Locator 对象放到我们自己的 Locator 对象中
        this.locator = locator;
     }

    public void startDocument() throws SAXException{
        System.out.println("文件分析开始->");
    }

    public void endDocument()throws SAXException{
        table.put(book.getTitle(),book);
        System.out.println("<-文件分析结束");
    }

    public void startPrefixMapping (String prefix, String uri) throws SAXException{
    System.out.println( "命名空间对应开始->" + "第" + locator.getLineNumber() + "行" + "\n\t 命名
空间前导符: " + prefix + "\n\t 相对应的统一资源标识符: " + uri);
        }

    public void endPrefixMapping (String prefix) throws SAXException{
        System.out.println("<-命名空间对应结束" );
        }

    public void startElement (String namespaceURI, String localName, String qName, Attributes
atts) throws SAXException{
        System.out.print("元素开始: " + localName);
        if(namespaceURI.equals(" ")) {
          namespaceURI = "没有命名空间";
        }
        System.out.println(", 命名空间: " + namespaceURI +", 限定名: " + qName + ".");
        //打印属性
        for(int i=0; i<atts.getLength(); i++)
         System.out.println("\t 属性名称: " + atts.getLocalName(i) +
                " = " + atts.getValue(i) + ".");
        if(localName.equals("book")){
          if(book!=null){
                table.put(book.getTitle(),book);
          }
          book = new BookBean();
        }
        }
```

```
        public void endElement (String namespaceURI, String localName,String qName) throws SAXException{
            System.out.print("元素结束: " + localName);
            if(namespaceURI.equals(" ")) {
                namespaceURI = "没有命名空间";
            }
 System.out.println(", 命名空间: " + namespaceURI +", 限定名: " + qName + ".");
            if(localName.equals("title")){
              book.setTitle(currentvalue);
            }
            if(localName.equals("url")){
              book.setUrl(currentvalue);
            }
            if(localName.equals("author")){
              book.setAuthor(currentvalue);
            }
            if(localName.equals("day")){
              book.setDay(this.currentvalue);
            }
            if(localName.equals("month")){
              book.setMonth(this.currentvalue);
            }
            if(localName.equals("year")){
              book.setYear(this.currentvalue);
            }
            if(localName.equals("description")){
              book.setDescription(this.currentvalue);
            }
          }

        public void characters (char ch[], int start, int length) throws SAXException{
            String charData = new String(ch, start, length);
            System.out.println("字符数据: \" " + charData + "\" ");
            currentvalue = charData;
            }

        public void ignorableWhitespace (char ch[], int start, int length) throws SAXException{
            String whiteSpace = new String(ch, start, length);
            System.out.println("空格符: \" " + whiteSpace + "\" ");
            }

        public void processingInstruction (String target, String data) throws SAXException{
            System.out.println("处理命令    目标(target):" + target+ "  和其数据(data):" + data);
            }

        public void skippedEntity (String name)throws SAXException{
            System.out.println("忽略的实体: " + name);
            }
        }
```

SAXHandlerBean 在解析文件之前，使用工厂利用一个指定的类名创建一个解析器：

```
            XMLReader   parser   =XMLReaderFactory.createXMLReader("org.apache.xerces.
parsers.SAXParser");
```

然后分别注册内容处理器和错误处理器：

```
    //注册自己设计的内容处理器
    parser.setContentHandler(this);
    //注册错误处理器
    parser.setErrorHandler(new SAXErrorHandler());
```

注意： 这里的内容处理器就是这个对象本身，因为这个类扩展了 ContentHandler 接口，具有内容处理的能力。

最后分析文件的内容：

```
    //分析文件
    parser.parse(uri);
```

当解析器在分析文件时，会遇到不同的元素或者属性，当遇到一个元素的开始或结束标志时都会触发一个事件，并由内容处理器中相应的方法进行处理。

在这个内容处理器类中，把所有的 book 元素集合为一个 book 对象，然后把所有的 book 对象存储在一个 Hashtable 对象中，并提供了两个方法用于获取这个 Hashtable 对象，分别用于不同的环境：

```
    public void getTable(HttpServletRequest request){
        request.getSession().setAttribute("books",table);
    }
    public Hashtable getTable(){
        return this.table;
    }
```

其中第一种方法只是把这个对象存储到 Session 对象中，如果需要使用这个对象，还需要从 Session 对象中获取。

3．SAXErrorHandler.java 文件

在上面的解析器中注册了一个内容处理器和一个错误处理器，内容处理器当然就是对 XML 文件的内容进行处理，但当在处理的过程中遇到错误时就需要错误处理器来处理。

SAX 的错误分为 3 个等级：error、warning 和 fatalError。任何错误处理器都要实现 ErrorHandler 接口，ErrorHandler 接口中提供了处理这 3 类错误的方法，任何错误处理器都必须实现这些方法。下面是错误处理器 SAXErrorHandler.java 的源代码：

```
package cn.ac.ict;

import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class SAXErrorHandler implements ErrorHandler {

    public SAXErrorHandler() {
```

```
            super();
            // TODO Auto-generated constructor stub
        }

        public void warning (SAXParseException e)throws SAXException {
            System.out.println(
                        "----分析警告----\n" +
                        "     发生位置:\t\t 第" + e.getLineNumber() + "行, " +
                        "     第" + e.getColumnNumber() + "字\n" +
                        "     系统标识符:\t" + e.getSystemId() + "\n" +
                        "     公用标识符:\t" + e.getPublicId() + "\n" +
                        "     错误消息:\t\t" + e.getMessage()
            );
            throw new SAXException("遇到警告.", e);
            }

        public void error (SAXParseException e)throws SAXException{
            System.out.println(
                        "----可恢复的分析错误----\n" +
                        "     发生位置:\t\t 第" + e.getLineNumber() + "行, " +
                        "     第" + e.getColumnNumber() + "字\n" +
                        "     系统标识符:\t" + e.getSystemId() + "\n" +
                        "     公用标识符:\t" + e.getPublicId() + "\n" +
                        "     错误消息:\t\t" + e.getMessage()
             );
             throw new SAXException("遇到可恢复错误.", e);
            }
        /**
         * *接受不可恢复的错误调用
         * *
         * */
        public void fatalError(SAXParseException e) throws SAXException{
            System.out.println(
                        "----不可恢复的分析错误----\n" +
                        "     发生位置:\t\t 第" + e.getLineNumber() + "行, " +
                        "     第" + e.getColumnNumber() + "字\n" +
                        "     系统标识符:\t" + e.getSystemId() + "\n" +
                        "     公用标识符:\t" + e.getPublicId() + "\n" +
                        "     错误消息:\t\t" + e.getMessage()
            );
            throw new SAXException("遇到不可恢复错误.", e);
            }
        }
```

4．SAXExample.jsp 文件

把上面的文件都准备好后，就可以编写使用这个解析器的 JSP 文件了，下面是 JSP 文件的内容：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import = "java.util.*"%>
```

```
<%@ page import = "cn.ac.ict.*"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
  <head>
    <title>SAX XML JSP</title>
  </head>
<body bgcolor="#FFFFFF">
<jsp:useBean id="saxparser" scope="session" class="cn.ac.ict.SAXHandlerBean" />

<%
//解析文件
      saxparser.parseXML("C:\\Tomcat 5.0\\webapps\\13\\books.xml");
//把解析结果存放到客户的 Session 中
      saxparser.getTable(request);
//从 Session 中获取解析结果
      Hashtable table = (Hashtable)session.getAttribute("books");
//循环迭代并输出解析结果
     Enumeration enum = table.keys();
         while(enum.hasMoreElements()){
            String booktitle = (String)enum.nextElement();
            BookBean book = (BookBean)table.get(booktitle);
            %>
书名：<%=book.getTitle()%><br>
URL:<%=book.getUrl()%><br>
作者：<%=book.getAuthor()%><br>
出版日期：<%=book.getDate()%><br>
<br><br>
<%}%>

</body>
</html>
```

在这个 JSP 文件中使用了解析器的 JavaBeans：

```
<jsp:useBean id="saxparser" scope="session" class="cn.ac.ict.SAXHandlerBean" />
```

然后指定让它解析一个文件，并把解析的结果存储到客户的会话对象中，之后，客户使用时就可以从会话对象中获取了：

```
            saxparser.parseXML("C:\\Tomcat 5.0\\webapps\\13\\books.xml");
            saxparser.getTable(request);
            Hashtable table = (Hashtable)session.getAttribute("books");
```

之后就可以对解析的结果进行操作了。

把需要使用 JavaBeans 编译后的字节码文件复制到 Web 应用的 WEB-INF\classes 目录下。

注意： 文件存放的目录层次和类的包应该一致。

把 JSP 文件复制到 Web 应用的根目录下，在浏览器地址栏中输入如下地址：http://localhost:8080/13/SAXExample.jsp，可以看到页面显示如图 13.4 所示。



图 13.4　SAX 实例

13.5　使用 XSLT 给 XML 定制样式

XSLT 的英文标准名称为 eXtensible Stylesheet Language Transformation。根据 W3C 的规范说明书（http://www.w3.org/TR/xslt），最早设计 XSLT 的用意是帮助 XML 文档（document）转换为其他文档。但随着不断的发展，XSLT 已不仅仅用于将 XML 转换为 HTML 或其他文本格式，更全面的定义应该是：XSLT 是一种用来转换 XML 文档结构的语言。

XSLT 转换的过程可以用图 13.5 表示。XSLT 处理器读取两个文件：XML 源文件和 XSLT 样式文件，然后转换成 HTML 或者其他的格式输出。XML 源文件的作用是提供数据，而 XSLT 样式文件则是规定了如何转换和转换后的格式。

例如如下的 XML 文件（hello.xml）：

```
<?xml version="1.0" encoding="iso-8859-1"?>
<greeting>Hello, world!</greeting>
```

使用浏览器打开这个文档，可以看到效果如图 13.6 所示。

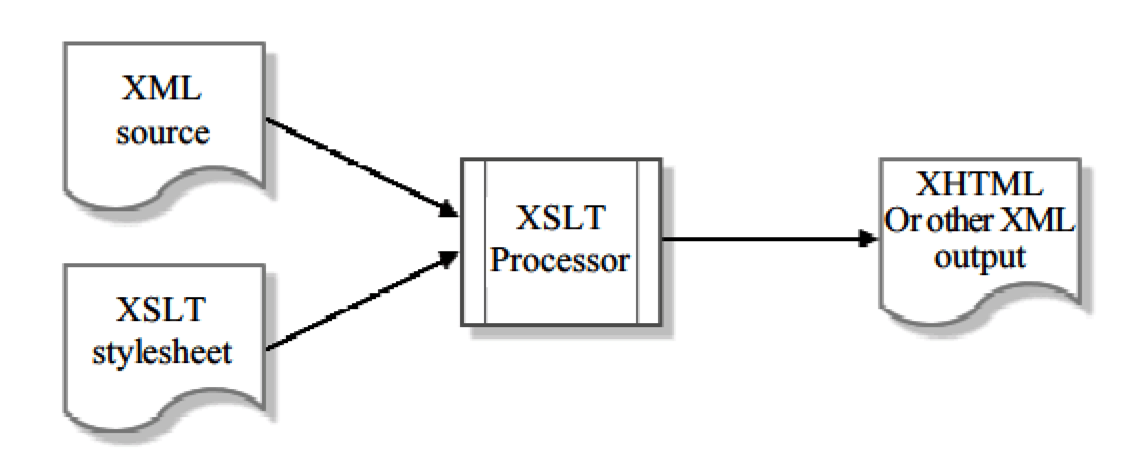


图 13.5　XSLT 转换的过程



图 13.6　使用浏览器查看 XML 文档

页面上只是传达了一个“Hello，world!”的信息，可屏幕上却显示了大量的无关字符，对于这个简单的例子，读者可能会看出它是什么意义，如果内容多了以后，就会发现阅读起来比较麻烦，虽然它显得已经很容易阅读了。

下面使用一个 XSLT 样式文件把这个文件转换为 HTML 格式的文件，让它以更容易阅读的格式显示出来，下面是这个 XSLT 样式文件的代码：

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```
<!--  声明使用的命名空间  -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<!---   定义一个模板 template   --->
<xsl:template match="/">
  <html>
    <head>
      <title>First XSLT example</title>
    </head>
    <body>
      <p><xsl:value-of select="greeting"/></p>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

在这个 XSLT 样式文件中，声明了它使用的命名空间，并添加了一个<xsl:template>元素，也就是 XSLT 的模板元素。XSLT 样式文件就是由一个个模板构成的，XSLT 处理器会使用各个模板对 XML 元素进行转换，然后把各个转换的结果组合起来，形成最终的新文档，其中<xsl:template>元素的 match 属性说明了这个模板可以被应用到的场合。

注意： 读者如果需要进一步地了解 XML 和 XSLT 的转换，可以参考其他的相关文档和书籍，这里由于篇幅所限，不再过多介绍。

有了 XML 文件和 XSLT 样式文件之后，如何使用它们进行转换呢？要实现 XSLT 转换还需要一个 XSLT 处理器，XSLT 处理器根据不同的需要也是有很多的。在这里，为了让读者能够看到 XSLT 转换的效果，就使用 IE 中默认使用的微软的一个 XSLT 处理器来进行转换。首先要修改 XML 文件，在这个文件中，声明使用的 XSLT 样式文件，修改后的代码如下：

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!---   声明使用的 XSLT 文件   --->
<?xml-stylesheet type="text/xsl" href="hello.xsl"?>
<greeting>Hello, world!</greeting>
```

微软的 XSLT 处理器 MSXML 现在的最高版本是 4.0 SP5，MSXML 是随 IE 浏览器一起发布的，如果 IE 的版本在 5.0 以上，就会有这个处理器，如果没有，读者可以到微软的官方网站下载，下载地址是：http://www.microsoft.com/downloads/details.aspx?FamilyID=4a3ad088-a893-4f0b-a932-5e024e74519f&DisplayLang=en。

使用浏览器打开修改过的 XML 文档（XML 文档要和 XSLT 样式文件放在同一个文件夹下），可以看到这时 XML 文档的显示效果好多了，如图 13.7 所示。



图 13.7　使用 MSXML 转换后的 XML 文档

上面的小例子，只是在客户端将 XML 文件按照 XSLT 文件的要求显示，下面介绍如何

在服务器端编程实现这样的转换。

13.5.1　建立 XML 文件

首先需要建立 XML 文件，XML 文件的建立比较简单，而且本章前面也有介绍，这里不作详细说明，下面是 XML 文件（books_2.xml）的内容：

```
<?xml version="1.0" encoding="gb2312"?>
<?xml-stylesheet type="text/xsl" href="bookhtml.xsl"?>
<books>
	<book id="1">
		<title url="http://java.sun.com">JSP 应用开发指南</title>
		<author>Author 001</author>
		<date>
			<day>23</day>
			<month>1</month>
			<year>2006</year>
		</date>
		<description>一本详尽介绍 JSP 技术的书</description>
	</book>

	<book id="2">
		<title url="http://ant.apache.org/">Ant 技术详解</title>
		<author>Author 002</author>
			<date>
				<day>23</day>
				<month>1</month>
				<year>2006</year>
			</date>
		<description>一本详尽介绍 Ant 技术的书</description>
	</book>

	<book id="3">
		<title url="http://www.apache.org/">Apache 技术详解</title>
		<author>Author 003</author>
		<date>
			<day>23</day>
			<month>1</month>
			<year>2006</year>
		</date>
		<description>一本详尽介绍 Apache 技术的书</description>
	</book>

</books>
```

13.5.2　建立 XSLT 文件

下面是使用的 XSLT 文件，具体关于 XSLT 文件的编写方法请读者参考其他的书籍。

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

```
  <xsl:template match="/">
  <HTML><TITLE>Using XSLT Style XML</TITLE><BODY>
  <xsl:apply-templates/>
  </BODY></HTML>
  </xsl:template>

<!--  列出所有的 books  -->
  <xsl:template match="books">
  <TABLE border="1">
  <TR><TD>Book  Title</TD><TD>Author</TD><TD>Publish  Date</TD><TD>Description</TD>
</TR>
  <xsl:apply-templates/>
  </TABLE>
  </xsl:template>

<!--  列出一个 book 的信息  -->
  <xsl:template match="book">
  <TR><TD><xsl:apply-templates select="title"/></TD><TD><xsl:value-of select="author"/></TD>
<TD><xsl:apply-templates  select="date"/></TD><TD><xsl:value-of  select="description"/></TD>
</TR>
  </xsl:template>

<!--  列出一个 book 的 title 的链接和文字信息  -->
  <xsl:template match="title">
  <A href="{@url}"><xsl:value-of select="."/></A>
  </xsl:template>

<!--  列出一个 book 的出版日期信息  -->
  <xsl:template match="date">
  <xsl:value-of select="year"/>-<xsl:value-of select="month"/>-<xsl:value-of select="day"/>
  </xsl:template>

</xsl:stylesheet>
```

13.5.3　建立源数据的对象

源数据的对象表示 XML 数据，这些数据用于创建最后的输出，把 XML 源数据和 XSLT 结合起来是使用 JAXP 完成的。

1．StreamSource

使用一个 XML 文件对象作为源数据时的代码如下：

```
StreamSource ss = new StreamSource(new File("Your_xml_file.xml"));
```

如果只是提供 XML 文件的名字，也可以使用字符串构建源数据的对象，代码如下：

```
StreamSource ss = new StreamSource(new StringReader(new String()));
```

2．DOMSource

DOMSource 表示 DOM 格式的源数据，建立 DOM 源数据对象的代码如下：

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.parse("Your_xml_file.xml");
DOMSource ds = new DOMSource(d);
```

3．SAXSource

SAXSource 表示 SAX 格式的源数据，建立 SAX 源数据对象的代码如下：

```
SAXSource ss = new SAXSource(new InputSource(new FileReader("Your_xml_file.xml")));
```

13.5.4　建立结果数据的对象

结果数据的对象是 JAXP 转换的输出结果，结果不一定是 XML 格式的，在 JAXP 中，有 3 个实现 Result 接口的对象：StreamResult、DOMResult 和 SAXResult。

1．StreamResult

StreamResult 是流的转换结果，在把结果写到文件或者 JSP 输出流时使用 StreamResult 比较多。下面是把结果输出到一个文件的例子：

```
StreamResult sr = new StreamResult(new File("example.out"));
```

当把结果输出到 JSP 的输出流中，代码如下：

```
StreamResult sr = new StreamResult(out);
```

2．DOMResult

DOMResult 保存 DOM 树型对象的转换结果，这个对象可以继续使用 Document 对象进行操作，当在应用之间传递 Document 对象时比较有用，此时的代码如下：

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.newDocument();
DOMResult dr = new DOMResult(d);
```

3．SAXResult

SAXResult 是实现 ContentHandler 接口的对象处理 SAX 事件的结果，下面是一段简单的代码：

```
SAXResult sr = new SAXResult(new DefaultHandler());
```

13.5.5　转换数据

通过上面几步的介绍，读者对建立一个简单的 JAXP 转换的过程应该有足够的了解了，要进行转换，使用 TransformerFactory 生成一个转换器是必不可少的，在本例中使用 Stream

Source 表示 XSLT 文件，下面是 JSP 文件（transform.jsp）的代码：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import="javax.xml.parsers.*,org.w3c.dom.*,javax.xml.transform.*,
javax.xml.transform.stream.*,java.io.*"%>
<%
//XML 文件
StreamSource xml = new StreamSource(new File("c:/xml/links.xml"));
//XSLT 文件
StreamSource xsl = new StreamSource(new File("c:/xml/links_html.xsl"));
//转换器工厂
TransformerFactory tFactory = TransformerFactory.newInstance();
//获取一个转换器
Transformer transformer = tFactory.newTransformer(xsl);
//得到结果，并输出到 JSP 页面
StreamResult result = new StreamResult(out);
//执行转换
transformer.transform(xml, result);
%>
```

在输出结果时，使用了 JSP 的隐含对象，将结果输出到 JSP 页面。

```
//得到结果，并输出到 JSP 页面
StreamResult result = new StreamResult(out);
```

把这个 JSP 文件发布到本章的 Web 应用中，在浏览器地址栏中输入如下地址：http://localhost:8080/13/transform.jsp，可以看到页面显示如图 13.8 所示。



图 13.8　使用 XSLT 文件格式化 XML 文件

13.6　小　　结

在本章中介绍了很多关于在 JSP 中使用 XML 文件的技术，随着 XML 的流行，JSP 和 XML 技术的结合将更加紧密，事实上，现在完全可以按照 XML 的格式书写 JSP 文件，读者如果感兴趣，可以参考相关书籍。

第 14 章　使用 Servlet 过滤器和监听器

Servlet API 很早就已成为企业应用开发的重要工具，而 Servlet 2.3 版中新增的过滤器和监听器功能则是对 J2EE 体系的一个补充，过滤器使得 Servlet 开发者能够在请求到达 Servlet 之前截取请求，在 Servlet 处理请求之后修改应答；而 Servlet 监听器可以监听客户端的请求、服务端的操作，通过监听器，可以自动激发一些操作，如监听应用的启动和停止。本章将介绍 Servlet 过滤器和监听器的相关知识以及在实际开发中如何使用这些技术。

14.1　Servlet 过滤器简介

Servlet 过滤器是小型的 Web 组件，它能拦截请求和响应，以便查看、提取或以某种方式操作正在客户机和服务器之间交换的数据。过滤器通常封装了一些功能的 Web 组件，这些功能虽然很重要，但对于处理客户机请求或发送响应来说不是决定性的。典型的例子包括记录关于请求和响应的数据、处理安全协议、管理会话属性等。

Servlet 过滤器介于与之相关的 Servlet 或 JSP 页面和客户之间，某个资源一旦与某个 Servlet 过滤器相关联，则对这个资源的任何请求或这个资源的响应都要经过与之关联的 Servlet 过滤器再调用或返回。

注意： 过滤器只能在与 Servlet 规范 2.3 版（含以上）兼容的服务器上运行。如果 Web 应用必须使用旧版服务器，就不能使用过滤器。

Servlet 过滤器接受请求并响应对象，然后过滤器会检查请求对象，决定将该请求转发给链中的下一个组件，或者中止该请求并直接向客户机发回一个响应。如果请求被转发了，它将被传递给链中的下一个资源（另一个过滤器、Servlet 或 JSP 页面）。在这个请求通过过滤器链并被服务器处理之后，一个响应将以相反的顺序通过该链发送回去。这样就给每个过滤器都提供了根据需要处理响应对象的机会。

一个过滤器可以被关联到任意多个资源，一个资源也可以关联到任意多个过滤器。例如在图 14.1 中有如下的关联关系。

- F1 同时被关联到 S1、S2、S3。
- F1、F2、F3 都和 S2 关联。

关联到同一个资源的过滤器形成一个过滤器链。例如对 S2 访问时，F1、F2、F3 就形成了一个过滤器链，客户要访问资源 S2，就要经过 F1 过滤器，然后是 F2、F3，最后才是要访问的资源，如果在经过上面这 3 个过滤器时出现了错误或者被禁止访问，客户的请求就无法到达资源 S2，在 Servlet 过滤器中，这个过滤器传递的过程是通过 FilterChain.dofilter()

方法实现的，当过滤器链到达末尾时，这个方法调用请求的资源。建立一个过滤器涉及下列 4 个步骤。

（1）建立一个实现 Filter 接口的类。这个类需要 3 个方法，分别是：doFilter、init 和 destroy。doFilter 方法包含主要的过滤代码（见第（2）步），在 init 方法中进行一些初始化的设置，而 destroy 方法进行清除过滤器占用的资源。

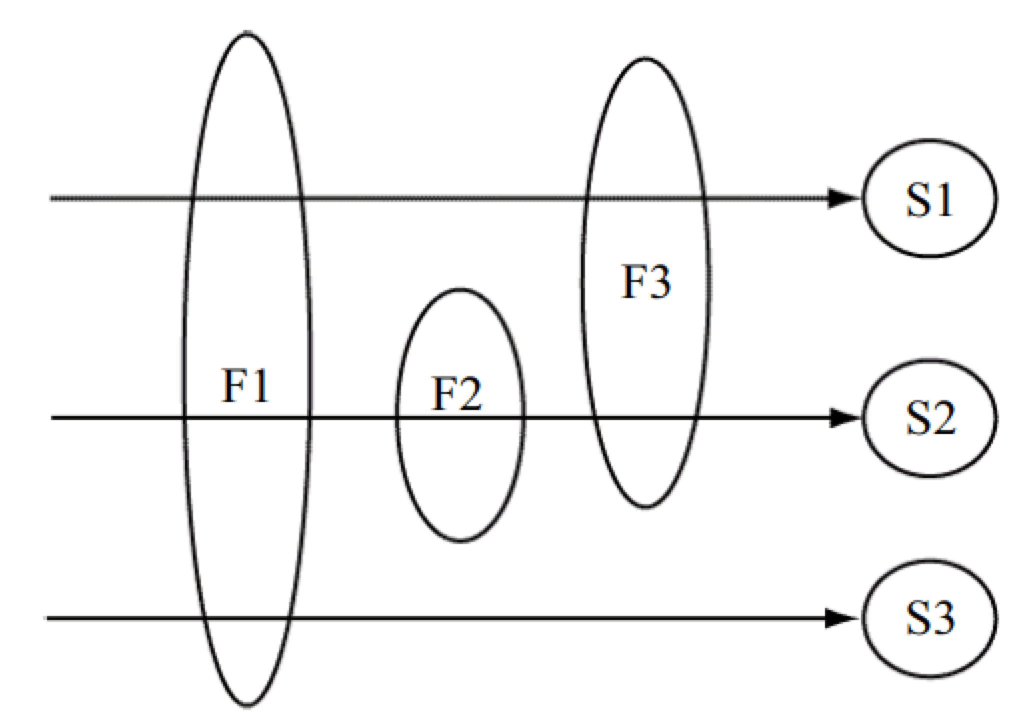


图 14.1　Web 资源与过滤器相关联

（2）在 doFilter 方法中放入过滤行为。doFilter 方法的第一个参数为 ServletRequest 对象。此对象给过滤器提供了对进入的信息（包括表单数据、Cookie 和 HTTP 请求头）的完全访问。第二个参数为 ServletResponse，通常在简单的过滤器中忽略此参数。最后一个参数为 FilterChain，如下一步所述，此参数用来调用过滤器链中下一个过滤器或者在到达过滤器末尾时调用 Servlet 或 JSP 页。

（3）调用 FilterChain 对象的 doFilter 方法。Filter 接口的 doFilter 方法把一个 FilterChain 对象作为它的一个参数。在调用此对象的 doFilter 方法时，激活下一个相关的过滤器。如果没有另一个过滤器与 Servlet 或 JSP 页面关联，则 Servlet 或 JSP 页面被激活。

（4）对相应的 Servlet 和 JSP 页面注册过滤器。在部署描述符文件（web.xml）中使用 filter 和 filter-mapping 元素对需要进行过滤的资源进行配置。

14.2　实现一个 Servlet 过滤器

Servlet 过滤器 API 包含 3 个简单的接口，它们在 javax.servlet 包中。这 3 个接口分别是 Filter、FilterChain 和 FilterConfig。从编程的角度看，过滤器类将实现 Filter 接口，然后使用这个过滤器类中的 FilterChain 和 FilterConfig 接口。该过滤器类的一个引用将传递给 FilterChain 对象，以允许过滤器把控制权传递给过滤器链中的下一个过滤器或者资源。FilterConfig 对象将由容器提供给过滤器，以允许访问该过滤器的初始化数据。

14.2.1　编写实现类的程序

下面介绍一个简单的 Servlet 过滤器的例子，在这个例子中，获取一个 Web 服务器给客户提供服务需要的时间，也就是响应时间。

```
package cn.ac.ict.Filter;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.Date;
```

```
import javax.servlet.*;

public class ResponseTimeFilter implements Filter {
        private FilterConfig filterConfig = null;
        //初始化函数
        public void init(FilterConfig config) throws ServletException {
                this.filterConfig = config;
        }

        public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
                Date startTime, endTime;
                double totalTime;
                startTime = new Date();
              StringWriter sw = new StringWriter();
              PrintWriter writer = new PrintWriter(sw);
              writer.println();
              writer.println("ResponseTimeFilter Before call chain.doFilter");

              // 把请求转发给过滤器链中下一个资源
              chain.doFilter(request, response);
              //下面是资源的响应时间
              //-------------
              //下面计算开始时间和结束时间的差，也就是响应时间
              endTime = new Date();
              totalTime = endTime.getTime() - startTime.getTime();
              //将毫秒时间转换为以秒为单位
              totalTime = totalTime / 1000;

              writer.println("ResponseTimeFilter Before call chain.doFilter");
              writer.println("===============");
              writer.println("Total elapsed time is: " + totalTime + " seconds." );
              writer.println("===============");
              //把信息输出到日志中
              filterConfig.getServletContext().log(sw.getBuffer().toString());
              writer.flush();
        }

        public void destroy() {
                this.filterConfig = null;
        }

        public FilterConfig getFilterConfig() {
                return null;
        }

        public void setFilterConfig(FilterConfig config) {
        }
}
```

当容器第一次加载该过滤器时，init()方法将被调用。该类在这个方法中包含了一个指向 FilterConfig 对象的引用。

过滤器的大多数时间都消耗在执行过滤操作上，doFilter()方法被容器调用，同时传入分别指向这个请求/响应链中的 ServletRequest、ServletResponse 和 FilterChain 对象的引用。然后过滤器就有机会处理请求，将处理任务传递给链中的下一个资源（通过调用 FilterChain 对象引用上的 doFilter()方法），之后在处理控制权返回该过滤器时处理响应。

在本实例中，过滤器接受到客户的请求后，记录当前的系统时间，然后把请求转发给过滤器链中其他资源，在得到响应后，再记录得到响应的时间，这个差值也就是响应时间了：

```
startTime = new Date();
StringWriter sw = new StringWriter();
PrintWriter writer = new PrintWriter(sw);
writer.println();
writer.println("ResponseTimeFilter Before call chain.doFilter");

// 把请求转发给过滤器链中下一个资源
chain.doFilter(request, response);
//下面是资源的响应时间
//-------------
//下面计算开始时间和结束时间的差，也就是响应时间
endTime = new Date();
totalTime = endTime.getTime() - startTime.getTime();
//将毫秒时间转换为以秒为单位
totalTime = totalTime / 1000;
```

最后把构造相关的信息输出到日志中：

```
writer.println("ResponseTimeFilter Before call chain.doFilter");
writer.println("===============");
writer.println("Total elapsed time is: " + totalTime + " seconds." );
writer.println("===============");
//把信息输出到日志中
filterConfig.getServletContext().log(sw.getBuffer().toString());
writer.flush();
```

14.2.2　配置发布 Servlet 过滤器

发布 Servlet 过滤器时，必须在 web.xml 文件中加入<filter>和<filter-mapping>元素，<filter>元素用来定义一个过滤器，如本例中使用如下代码：

```
<filter>
    <filter-name>Request Response Time</filter-name>
<filter-class>cn.ac.ict.Filter.ResponseTimeFilter</filter-class>
</filter>
```

以上代码中<filter-name>指定过滤器的名字，<filter-class>指定过滤器的完整类名，也可以在<filter>元素中内嵌<init-param>元素来为过滤器提供初始化参数；<filter-mapping>元素

用于将过滤器和 URL 关联，本例中的代码如下：

```
<filter-mapping>
    <filter-name>Request Response Time</filter-name>
    <url-pattern>/ShowCounter.jsp</url-pattern>
</filter-mapping>
```

当客户请求的 URL 和<url-pattern>指定的 URL 相匹配时，就会触发 ResponseTimeFilter 过滤器，这里的<filter-name>元素的值和<filter>元素中的<filter-name>值应该是相同的。

注意： 如果希望 Servlet 过滤器能过滤所有的客户请求，可以把<url-pattern>的值设置为/*。

准备好上面的文件后，读者可以按照如下步骤发布这个 Servlet 过滤器：

（1）编译 ResponseTimeFilter 这个类，编译时，需要把 Java Servlet API 的包 servlet-api.jar 放到类路径中，编译后的类放到本章 Web 应用的 WEB-INF\classes 目录下，并且目录结构要和包的结构一致。

（2）在 web.xml 文件中按照上面说明的方法配置这个过滤器。

（3）为了观察这个过滤器的日志，应该确保在 server.xml 文件的 localhost 对应的 host 元素中配置了如下的 logger 元素：

```
<Logger className="org.apache.catalina.logger.FileLogger"
        directory="logs"   prefix="localhost_log." suffix=".txt"
    timestamp="true"/>
```

默认情况下是配置好的。

（4）启动 Tomcat，在浏览器地址栏中输入如下地址：http://localhost:8080/14/ShowCounter.jsp，这时，在<TOMCAT_HOME>\logs\ localhost_log.2006-02-12.txt 文件中会生成如下的日志信息：

```
2006-02-12 23:03:42 StandardContext[/14]
ResponseTimeFilter Before call chain.doFilter
ResponseTimeFilter Before call chain.doFilter
===============
Total elapsed time is: 0.0 seconds.
===============
```

注意： 本例中访问的 ShowCounter.jsp 页面将在本章的监听器部分介绍，如果读者不想跳跃着查看这个文件，可以尝试修改过滤器关联的 URL，使得它可以过滤读者指定的资源。

14.3　ServletRequest 和 ServletResponse 的包装类

Servlet 过滤器能够对客户的请求和响应进行包装，并根据自定义的行为对请求和响应进行修改，这就是通过使用 ServletRequest 和 ServletResponse 的包装类——HttpServletRequestWrapper 和 HttpServletResponseWrapper 类来实现的，这两个类提供了所有 request 和 response 方法的默认实现，并且默认代理了所有对原有 request 和 response 的调用。这意味着要改变

一个方法的行为只需要从封装类继承并重新实现一个方法即可。

- ServletRequestWrapper 类是 ServletRequest 的包装类，它实现了 ServletRequest 接口，并对其方法提供了默认实现。
- ServletResponseWrapper 类是 ServletResponse 的包装类，它实现了 ServletResponse 接口，并对其方法提供了默认实现。

14.4　用 Servlet 过滤器过滤文本信息

在 14.3 节中简单介绍了 ServletRequest 和 ServletResponse 的包装类，它们在过滤器设计中还是会经常用到的，在本节中介绍一个使用 HttpServletResponseWrapper 类的例子，客户输入的内容会被检查，如果有不允许出现的信息时，就会被过滤掉，而被替换成其他的字符。

在这个例子中，客户输入用户名并发布自己的留言信息，然后提交给服务器，服务器再把这些信息反馈给客户，如果客户的留言中有指定的字符串就会调用过滤器将这个字符串替换掉。

14.4.1　输出流管理类

在 Servlet 输出时可以选择不同的输出流，将它们集合在一起，就可以很容易地对这些流进行管理，使用时也会更方便，下面是这个管理类的代码（ByteArrayPrintWriter.java）：

```
package cn.ac.ict.Filter;
import java.io.ByteArrayOutputStream;
import java.io.PrintWriter;
import javax.servlet.ServletOutputStream;

public class ByteArrayPrintWriter {
      private ByteArrayOutputStream baos = new ByteArrayOutputStream();
      private PrintWriter pw = new PrintWriter(baos);
      private ServletOutputStream sos = new ByteArrayServletStream(baos);

      public ByteArrayPrintWriter(ByteArrayOutputStream aos){
       this.baos = aos;
      }
      //一个为空的构造方法
       public ByteArrayPrintWriter() {
       }
//使用字符输出流时调用获取输出流
       public PrintWriter getWriter() {
         return pw;
      }
//使用字节输出流时调用获取输出流
      public ServletOutputStream getStream() {
```

```
        return sos;
    }
    byte[] toByteArray() {
        return baos.toByteArray();
    }
}
```

在这个类中定义了 3 个对象，一个是使用字符流输出的，两个是用字节流输出的，但是它们都最终套接到类型为 ByteArrayOutputStream 的对象上，这个对象为它们提供原始输出数据。

14.4.2　编写响应包装类

这个例子中需要把服务器的响应进行检查，必要时可能会修改其内容，这时就需要使用响应包装类，把响应先拦截下来处理后再返回给客户，下面是响应包装类的代码：

```
package cn.ac.ict.Filter;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServletResponseWrapper;

public class TextModifierResponseWrapper extends HttpServletResponseWrapper {
      private HttpServletResponse resp;
      private ByteArrayPrintWriter bapw;
      public   TextModifierResponseWrapper(HttpServletResponse   response,ByteArrayPrintWriter
pw0) throws IOException {
            super((HttpServletResponse) response);
            this.resp = response;
            this.bapw = pw0;
      }

     public PrintWriter getWriter() {
          return bapw.getWriter();
       }
     public ServletOutputStream getOutputStream() {
          return bapw.getStream();
       }

     public void setContentType(String type) {
          if (type.equals("text/xml")) {
            resp.setContentType("text/html");
             } else {
                  resp.setContentType(type);
          }
       }

}
```

关于响应包装类前面也有介绍，值得注意的一点就是需要重写在响应类可能调用的方法，例如，响应（HttpServletResponse 的对象）既有可能调用 getWriter 来使用字符输出流输出数据，也有可能调用 getOutputStream 方法来使用字节输出流输出数据，因此这两种方法是很有必要的，另外，在本例中还设置了响应的类型，因此重写 setContentType 方法也是很有必要的。

14.4.3　编写 Servlet 过滤器

前面编写了需要使用的辅助类，下面就可以开发进行实际过滤功能的文本过滤器了，其代码（TextModifierFilter.java）如下：

```
package cn.ac.ict.Filter;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class TextModifierFilter implements Filter {
        private FilterConfig filterConfig;
        private String searchStr;
        private String replaceStr;
        public void init(FilterConfig config) throws ServletException {
                this.filterConfig = config;
//初始化要查找的字符串和替换后的字符串的值
                this.searchStr = "search";
                this.replaceStr = "replace";

        }
        public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
                HttpServletRequest hsr = (HttpServletRequest)request;
                final HttpServletResponse resp = (HttpServletResponse)response;
                  final ByteArrayPrintWriter pw = new ByteArrayPrintWriter();
             //构造响应包装类
                TextModifierResponseWrapper tmrw = new TextModifierResponseWrapper(resp,pw);
                filterConfig.getServletContext().log("TextModifierFilter:Before call chain.doFilter");
             //将请求转发给 Servlet，并获取响应
                chain.doFilter(request,tmrw);
             //下面的代码检查响应并对需要替换的字符进行替换
             //将响应转化为 byte 数组
               byte[] bytes = pw.toByteArray();
               if (bytes == null || (bytes.length == 0)) {
```

```
                filterConfig.getServletContext().log("No content!");
               }
             //将其转换为字符串便于查找
                String content = new String(bytes);
                String searchcontent = (new String(bytes)).toLowerCase();
             //查找指定字符出现的第一个位置
                int endPos = searchcontent.indexOf(this.searchStr);
                String returnStr;
             //响应中包含这样一个字符串
                if(endPos!=-1){
                        String front_part_Str = content.substring(0,endPos);
                        returnStr = front_part_Str + "<font color=red>"+this.replaceStr+"</font>" +content.
substring(endPos + this.searchStr.length());
                }else{
             //响应中不包含这样一个字符串
                        returnStr = content;
                }
                resp.setContentLength(returnStr.length());
                resp.getOutputStream().write(returnStr.getBytes());
                filterConfig.getServletContext().log("TextModifierFilter:After call chain.doFilter");
        }

        public void destroy() {
                this.filterConfig = null;
        }

        public FilterConfig getFilterConfig() {
                return null;
        }

        public void setFilterConfig(FilterConfig config) {
        }
}
```

上述代码的主要功能集中在 doFilter 方法中，在这个方法中构造了一个响应包装器，当调用 chain.doFilter(request,tmrw);方法时，Servlet 的响应都已经在响应包装器对象 tmrw 中了，之后就可以调用相关的方法获取响应的内容，并进行适当的处理了。

```
//将响应转化为 byte 数组
               byte[] bytes = pw.toByteArray();
               if (bytes == null || (bytes.length == 0)) {
                filterConfig.getServletContext().log("No content!");
               }
//将其转换为字符串便于查找
                String content   = new String(bytes);
                String searchcontent = (new String(bytes)).toLowerCase();
//查找指定字符出现的第一个位置
                int endPos = searchcontent.indexOf(this.searchStr);

                String returnStr;
```

```
//响应中包含这样一个字符串
        if(endPos!=-1){
            String front_part_Str = content.substring(0,endPos);
            returnStr = front_part_Str + "<font color=red>"+this.replaceStr+"</font>" +content.
substring(endPos + this.searchStr.length());
        }else{
//响应中不包含这样一个字符串
            returnStr = content;
        }
        resp.setContentLength(returnStr.length());
        resp.getOutputStream().write(returnStr.getBytes());
```

14.4.4　编写 JSP 和 Servlet 文件

在前面的介绍中，已经开发完成了一个完整的文本过滤器的程序，在本节中开发演示过滤器工作过程的 JSP 和 Servlet 文件。JSP 文件代码如下：

```
<%@ page language="java" pageEncoding="GB2312" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
  <head>
    <title>User Message</title>
  <style type="text/css">
  <!--
  .unnamed1 {
      border: 2px solid;
  }
  -->
  </style>
  </head>
<body bgcolor="#FFFFFF">
<form method="get" action ="textModifier">
<table width="580" border="0" align="center">
<thead align="center"><font style="font-size:24px; color:#FF0000;">User Message</font></thead>
<tr><td colspan="2"></td></tr>
  <tr>
    <td class="unnamed1">UserName</td>
    <td class="unnamed1"><input type=text name = username></input></td>
  </tr>
  <tr>
    <td   class="unnamed1">Message:</td>
    <td class="unnamed1"><textarea name="content" cols="40" rows="5"></textarea></td>
  </tr>
  <tr>
    <td class="unnamed1" align="center"><input type = submit value="submit"></input></td>
    <td class="unnamed1" align="center"><input type="reset" value="reset"></td>
  </tr>
```

```
</table>

</form>

</body>
</html>
```

在这个 JSP 页面中提供用户输入信息的界面，界面的效果如图 14.2 所示。

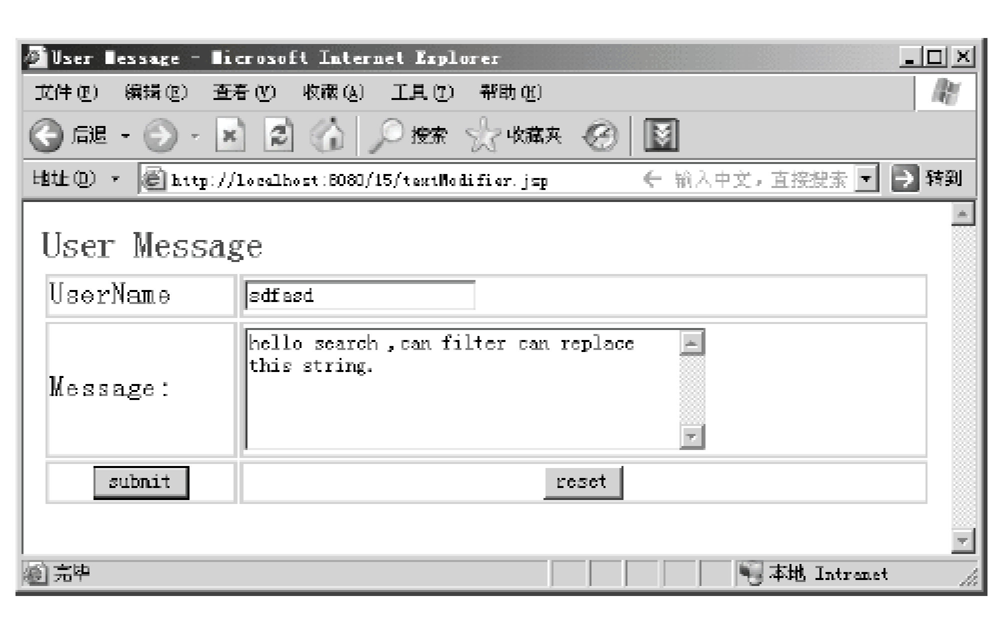


图 14.2　用户输入界面

下面是用户提交输入留言信息的响应 Servlet，其代码如下：

```
package cn.ac.ict.Servlet;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class TextModiferServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
//设置响应的类型
            response.setContentType("text/html;charset=gb2312");
            ServletOutputStream out = response.getOutputStream();
//获取请求参数
            String user = request.getParameter("username");
            String content = request.getParameter("content");
            String outstr = " ";
            if(user!=null){
                user = new String(user.getBytes("ISO8859-1"),"GB2312");
            }
            if(content!=null){
                content = new String(content.getBytes("ISO8859-1"),"GB2312");
            }
            if((user!=null)&&(content!=null)){
                 outstr="User["+user+"] say: ' "+content+" ' ";
            }
             out.println("\r\n");
             out.println("<!DOCTYPE HTML PUBLIC \"-//w3c//dtd html 4.0 transitional//en\">\r\n");
             out.println("<html>\r\n");
             out.println("<head>\r\n");
```

```
            out.println("<title>User Message</title>\r\n");
            out.println("<style type=\"text/css\">\r\n");
            out.println("<!--\r\n");
            out.println(".unnamed1 {\r\n");
            out.println("\tborder: 2px solid;\r\n");
            out.println("}\r\n");
            out.println("-->\r\n");
            out.println("</style>\r\n");
            out.println("</head>\r\n");
            out.println("<body bgcolor=\"#FFFFFF\">\r\n");
            out.println("<form method=\"get\" action =\"textModifier\">\r\n");
            out.println("<table width=\"580\" border=\"0\" align=\"center\">\r\n");
            out.println("<thead  align=\"center\"><font  style=\"font-size:24px;  color:#FF0000;\">
User Message</font></thead>\r\n");
          out.println("<tr><td colspan=\"2\">"+outstr+"</td></tr>\r\n");
            out.println(" <tr>\r\n");
            out.println(" <td class=\"unnamed1\">UserName</td>\r\n");
            out.println("  <td  class=\"unnamed1\"><input  type=text  name  =  username></input>
</td>\r\n");
            out.println(" </tr>\r\n");
            out.println(" <tr>\r\n");
            out.println(" <td class=\"unnamed1\">Message:</td>\r\n");
            out.println(" <td class=\"unnamed1\"><textarea name=\"content\" cols=\"40\" rows =\"5
\"></textarea></td>\r\n");
            out.println(" </tr>\r\n");
            out.println(" <tr>\r\n");
            out.println(" <td class=\"unnamed1\" align=\"center\"><input type=submit value= \"submit\">
</input></td>\r\n");
            out.println(" <td class=\"unnamed1\" align=\"center\"><input type=\"reset\" value=\"reset\">
</td>\r\n");
            out.println("   </tr>\r\n");
            out.println("</table>\r\n");
            out.println("\r\n");
            out.println("\r\n");
            out.println("\r\n");
            out.println("\r\n");
            out.println("</form>\r\n");
            out.println("\r\n");
            out.println("\r\n");
            out.println("</body>\r\n");
            out.println("</html>");
    }

    protected void doPost(HttpServletRequest request,
                HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }

}
```

在这个 Servlet 中，根据用户的留言信息构造一个消息再反馈给用户，而且保留了用户输入的各个组件。

14.4.5　发布运行 Web 应用

在把所有需要使用的文件都准备好后，读者可以按照如下步骤发布这个 Web 应用：

（1）编译输出流管理类、响应包装器和过滤器的 Java 程序，编译时需要把 servlet-api.jar 放到类路径中。编译完成后，把得到的字节码文件复制到 Web 应用的 WEB-INF\classes 目录下，存放位置与包的结构一致。

（2）在 Web 应用的 web.xml 文件中配置使用的 Servlet 和 Servlet 过滤器，也就是在 web.xml 文件中添加如下代码：

```
<!-- 配置 Servlet  -- >
   <servlet>
      <servlet-name>textModifier</servlet-name>
      <servlet-class>cn.ac.ict.Servlet.TextModiferServlet</servlet-class>
    </servlet>


    <servlet-mapping>
          <servlet-name>textModifier</servlet-name>
          <url-pattern>/textModifier</url-pattern>
    </servlet-mapping>
<!-- 配置 Servlet 过滤器  -- >
         <filter>
              <filter-name>Text Content Modifier</filter-name>

<filter-class>cn.ac.ict.Filter.TextModifierFilter</filter-class>
         </filter>

         <filter-mapping>
              <filter-name>Text Content Modifier</filter-name>
              <url-pattern>/textModifier</url-pattern>
         </filter-mapping>
```

（3）启动 Tomcat 或者重新加载 Web 应用，在浏览器地址栏中输入如下地址：http://localhost:8080/14/textModifier.jsp，并填写内容，显示页面如图 14.2 所示。

（4）可以看到用户的留言中包含了字符串 search。提交后可以看到页面显示如图 14.3 所示。

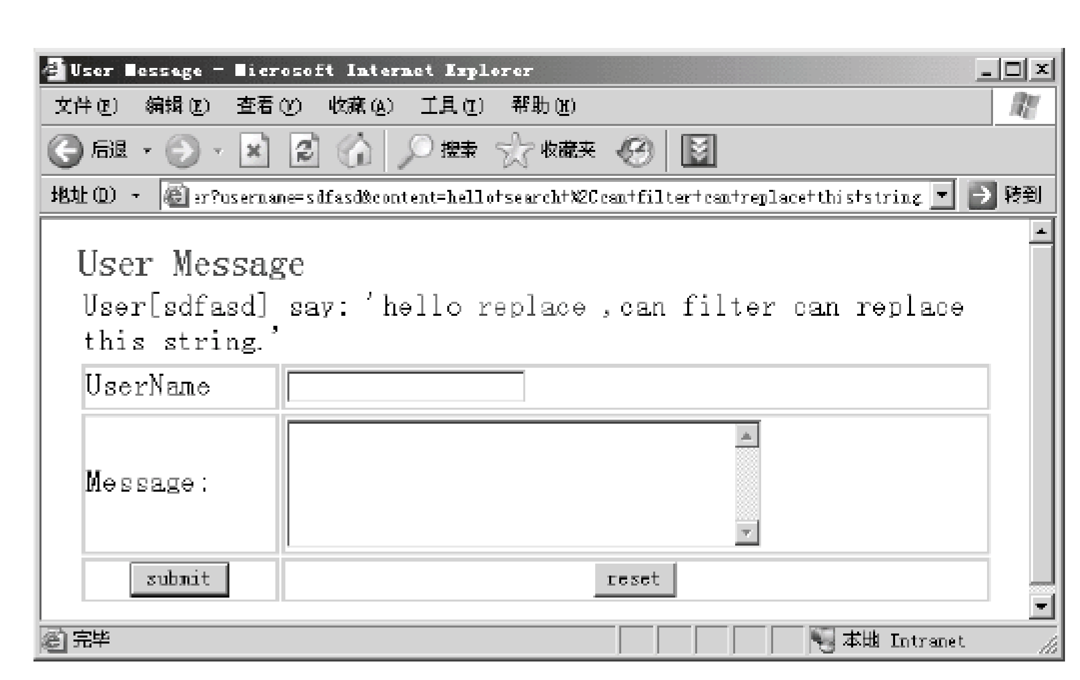


图 14.3　过滤效果

14.5　Servlet 监听器简介

Servlet 监听器与 Servlet 过滤器的很多特性是一致的，它的很多概念与 Java 中的监听器的概念也是一致的。它可以在特定事件发生时监听到，并根据其作出相应的反应。

Servlet 监听器是在 Servlet 2.3 版中引入的技术，它可以监听客户端的请求、服务端的操作等。Servlet 监听器是比较容易理解的。下面介绍 Servlet 监听器常用类和接口并举例讲述如何使用 Servlet 监听器。

14.5.1　监听服务器 ServletContext 对象

对 ServletContext 对象进行监听的接口有：ServletContextAttributeListener 和 ServletContextListener，它们可以分别用于监听 ServletContext 对象中属性的变化和 ServletContext 对象本身的变化。

下面对这两个接口作一简单介绍，首先介绍 ServletContextAttributeListener 接口。

实现 ServletContextAttributeListener 接口的监听器可以监听到 ServletContext 对象中属性的变化，它提供的方法有：

- void attributeAdded(ServletContextAttributeEvent scab)。
- void attributeRemoved(ServletContextAttributeEvent scab)。
- void attributeReplaced(ServletContextAttributeEvent scab)。

实现 ServletContextAttributeListener 接口可以监听对 ServletContext 属性的操作，当进行增加、删除、修改等操作时，都会调用相应的方法。

- 当在 ServletContext 增加一个属性时，激发 attributeAdded(ServletContextAttributeEvent scab)方法。
- 当在 ServletContext 删除一个属性时，激发 attributeRemoved(ServletContextAttributeEvent scab)方法。
- 当在 ServletContext 属性被重新设置时，激发 attributeReplaced(ServletContextAttributeEvent scab)方法。

下面介绍 ServletContextListener 接口，实现 ServletContextListener 接口的监听器可以监听 ServletContext 对象本身的变化，它提供的方法有：

- void contextDestroyed(ServletContextEvent sce)。
- void contextInitialized(ServletContextEvent sce)。

实现 ServletContextListener 接口可以监听 ServletContext 的操作。

- 当创建 ServletContext 时，激发 contextInitialized(ServletContextEvent sce)方法。
- 当销毁 ServletContext 时，激发 contextDestroyed(ServletContextEvent sce)方法。

14.5.2　监听客户会话

对客户会话进行监听的接口有：HttpSessionListener 接口、HttpSessionAttributeListener 接口、HttpSessionActivationListener 接口和 HttpSessionBindingListener 接口，它们可以分别用于监听 HttpSession 对象中属性的变化和 HttpSession 对象本身状态的变化。

下面对前 3 个接口作一简单介绍，首先介绍 HttpSessionAttributeListener 接口。

实现 HttpSessionAttributeListener 接口的监听器可以监听到 HttpSession 对象中属性的变化，它提供的方法有：

- void attributeAdded(HttpSessionBindingEvent se)。
- void attributeRemoved(HttpSessionBindingEvent se)。
- void attributeReplaced(HttpSessionBindingEvent se)。

实现 HttpSessionAttributeListener 接口可以监听 HttpSession 中属性的操作。

- 当在 Session 增加一个属性时，激发 attributeAdded(HttpSessionBindingEvent se)方法。
- 当在 Session 删除一个属性时，激发 attributeRemoved(HttpSessionBindingEvent se)方法。
- 当在 Session 属性被重新设置时，激发 attributeReplaced(HttpSessionBindingEvent se)方法。

下面介绍 HttpSessionListener 接口，实现 HttpSessionListener 接口的监听器可以监听 HttpSession 对象本身的创建和销毁，它提供的方法有：

- void sessionCreated(HttpSessionEvent se)。
- void sessionDestroyed(HttpSessionEvent se)。

实现 HttpSessionListener 接口的监听器可以监听 HttpSession 的操作。

- 当创建一个 Session 时，激发 sessionCreated(HttpSessionEvent se)方法。
- 当销毁一个 Session 时，激发 sessionDestroyed (HttpSessionEvent se)方法。

HttpSessionActivationListener 接口用于监听 HttpSession 对象的状态，它提供的方法有：

- void sessionDidActivate(HttpSessionEvent se)。
- void sessionWillPassivate(HttpSessionEvent se)。

实现 HttpSessionActivationListener 接口可以监听 HttpSession 对象是被激活还是钝化。

- 当激活一个 HttpSession 对象时激发 sessionDidActivate(HttpSessionEvent se)方法。
- 当钝化一个 HttpSession 对象时激发 sessionWillPassivate (HttpSessionEvent se)方法。

14.5.3　监听客户请求

对 ServletRequest 对象进行监听的接口有：ServletRequestAttributeListener 和 ServletRequestListener，这个是 Servlet 2.4 版本的新增监听器，它们可以分别用于监听 ServletRequest 对象中属性的变化和 ServletRequest 对象本身的变化。

下面对这两个接口作一简单介绍，首先介绍 ServletRequestAttributeListener 接口：

实现 ServletRequestAttributeListener 接口的监听器可以监听到 ServletRequest 对象中属性的变化，它提供的方法有：

- ❑ void attributeAdded(ServletRequestAttributeEvent scab)。
- ❑ void attributeRemoved(ServletRequestAttributeEvent scab)。
- ❑ void attributeReplaced(ServletRequestAttributeEvent scab)。

实现 ServletRequestAttributeListener 接口可以监听对 ServletRequest 属性的操作，当进行增加、删除、修改等操作时，都会调用相应的方法。

- ❑ 当在 ServletRequest 增加一个属性时，激发 attributeAdded(ServletRequestAttributeEvent scab)方法。
- ❑ 当在 ServletRequest 删除一个属性时，激发 attributeRemoved(ServletRequestAttributeEvent scab)方法。
- ❑ 当在 ServletRequest 属性被重新设置时，激发 attributeReplaced(ServletRequestAttributeEvent scab)方法。

下面介绍 ServletRequestListener 接口，实现 ServletRequestListener 接口的监听器可以监听 ServletRequest 对象本身的变化，它提供的方法有：

- ❑ void requestDestroyed(ServletRequestEvent sce)。
- ❑ void requestInitialized(ServletRequestEvent sce)。

实现 ServletRequestListener 接口可以监听 ServletRequest 的操作。

- ❑ 当创建 ServletRequest 时，激发 requestInitialized(ServletRequestEvent sce)方法。
- ❑ 当销毁 ServletRequest 时，激发 requestDestroyed(ServletRequestEvent sce)方法。

14.6　网站计数器实例——使用 Servlet 监听器

在本节介绍一个使用 Servlet 监听器对网站访问量进行统计的一个应用，在这个例子中使用了上面介绍的大部分接口，下面就通过分析这个例子介绍如何编写 Servlet 监听器。下面是这个监听器实现类的源代码：

```
package cn.ac.ict.Listener;

import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class CountHitListener implements HttpSessionListener,
        ServletContextAttributeListener, ServletRequestListener {
    private int count ,hit;
    public CountHitListener(){
        count = 0;
```

```
        hit = 0;
    }

    //创建一个会话时激发，将访问人数加 1
    public void sessionCreated(HttpSessionEvent arg0) {
        count++;
        setContext(arg0);

    }

     //设置 context 的属性，将计数器的值保存在 ServletContext 对象中
     //将会激发 attributeAdded 或者 attributeReplaced 方法（没有实现接口，这些方法在这个类中
不存在）
    private void setContext(HttpSessionEvent arg0) {
        arg0.getSession().getServletContext().setAttribute("Counter",new Integer(count));

    }
    //销毁一个会话时激发，将访问数减 1
    public void sessionDestroyed(HttpSessionEvent arg0) {
        count--;
        setContext(arg0);

    }

    //在 ServletContext 的对象中新添加一个属性时激发
    public void attributeAdded(ServletContextAttributeEvent arg0) {

    }

    //在 ServletContext 的对象中删除一个属性时激发
    public void attributeRemoved(ServletContextAttributeEvent arg0) {

    }

    //在 ServletContext 的对象中一个属性被替换时激发
    public void attributeReplaced(ServletContextAttributeEvent arg0) {

    }

    //一个请求结束时激发
    public void requestDestroyed(ServletRequestEvent arg0) {
    }

    //一个请求来临时激发
    public void requestInitialized(ServletRequestEvent arg0) {
        hit++;
        this.setContext(arg0);
    }
    //设置 ServletRequest 对象的属性
```

```
        private void setContext(ServletRequestEvent arg0) {
                arg0.getServletContext().setAttribute("Hiter",new Integer(hit));
        }
}
```

发布这个监听器时需要在 Web 应用的 web.xml 文件中添加如下代码：

```
        <listener>
<listener-class>cn.ac.ict.Listener.CountHitListener</listener-class>
       </listener>
```

下面编写一个简单的查看访问量的 JSP 页面，代码如下：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import ="java.lang.*"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
   <head>
      <title>显示网站在线用户数和页面点击量</title>
   </head>
<body bgcolor="#FFFFFF">
<%
   int count =((Integer) application.getAttribute("Counter")).intValue();
   int hit = ((Integer) application.getAttribute("Hiter")).intValue();
%>
      本站总线人数为：<%=count%>
      页面的点击量为：<%=hit%>
</body>
</html>
```

下面介绍这个监听器监测在线用户数量时的执行过程，如图 14.4 所示。

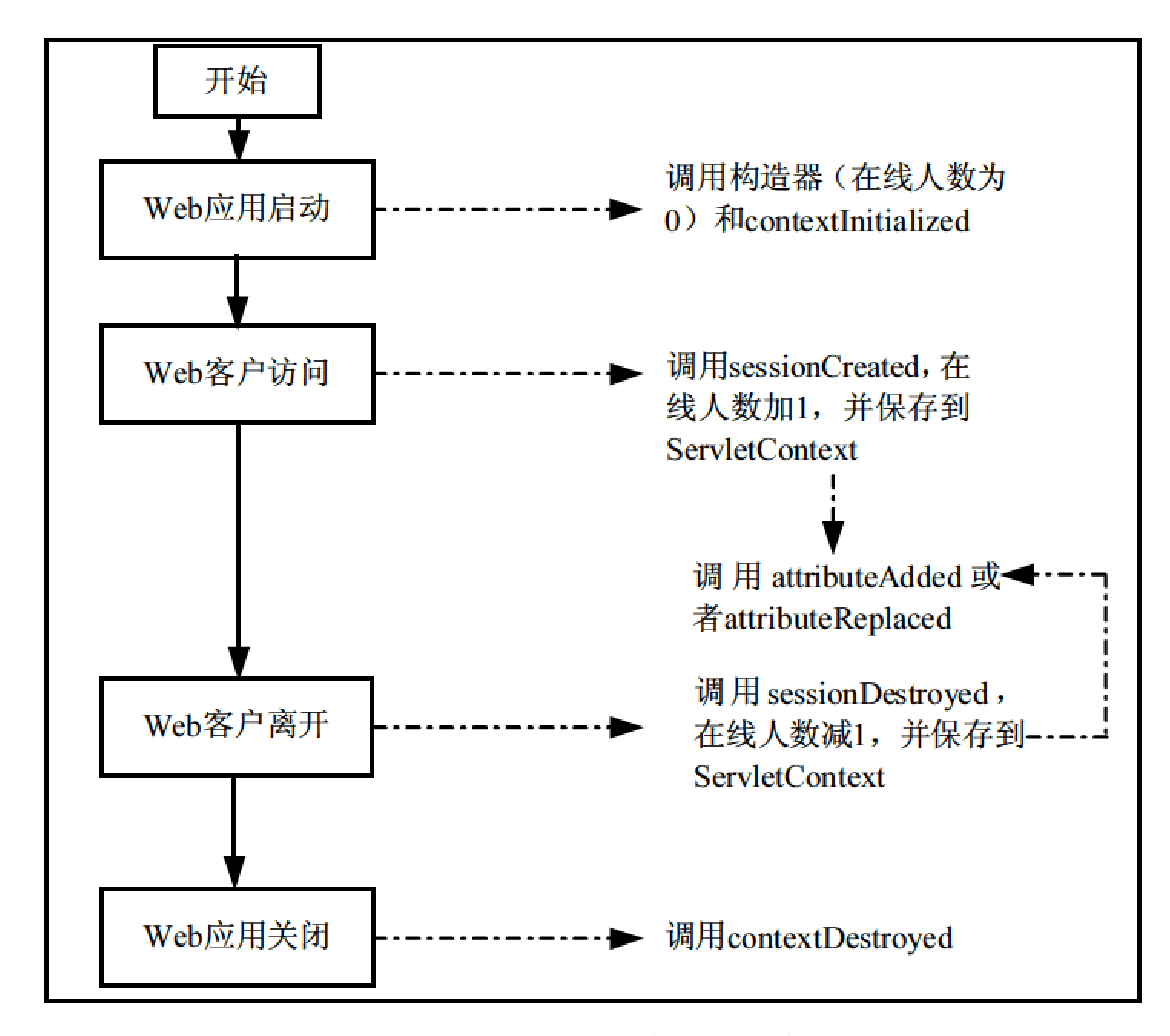


图 14.4　在线人数统计分析

从图 14.4 可以看到，Servlet 监听器提供服务的过程如下：

（1）当 Web 应用启动时，ServletContext 第一次建立，这时，调用 Listener 的构造器和 contextInitialized 对这个程序进行一些初始化设置。

（2）Web 应用启动后，就可以等待 Web 客户的访问了。当一个新的客户来到时，它会新开始一个会话，这时调用 sessionCreated 方法，将在线人数加 1，并把这个数值对象作为属性加入到 ServletContext 中。

（3）上述操作造成了 ServletContext 属性的变更，如果是第一个客户，就会调用 attributeAdded 方法，否则就会调用 attributeReplaced 方法，在这些方法中是不需要做任何事情的，所以程序中这些方法都不需要实现，因此都为空。

（4）客户离开这个 Web 应用时，就结束了一个会话，监听器会调用 sessionDestroyed 方法，把在线人数减 1，并把更改后的数值存储到 ServletContext 对象中。

（5）上述操作也造成了 ServletContext 属性的变更，会调用 attributeReplaced 方法。

（6）当 Web 应用被关闭或者 Tomcat 服务器被关闭时，一个 ServletContext 对象被销毁，这时监听器调用 contextDestroyed 方法。

将这个 Web 应用发布后，在浏览器地址栏中输入如下地址：http://localhost:8080/14/ShowCounter.jsp，可能的页面显示如图 14.5 所示。

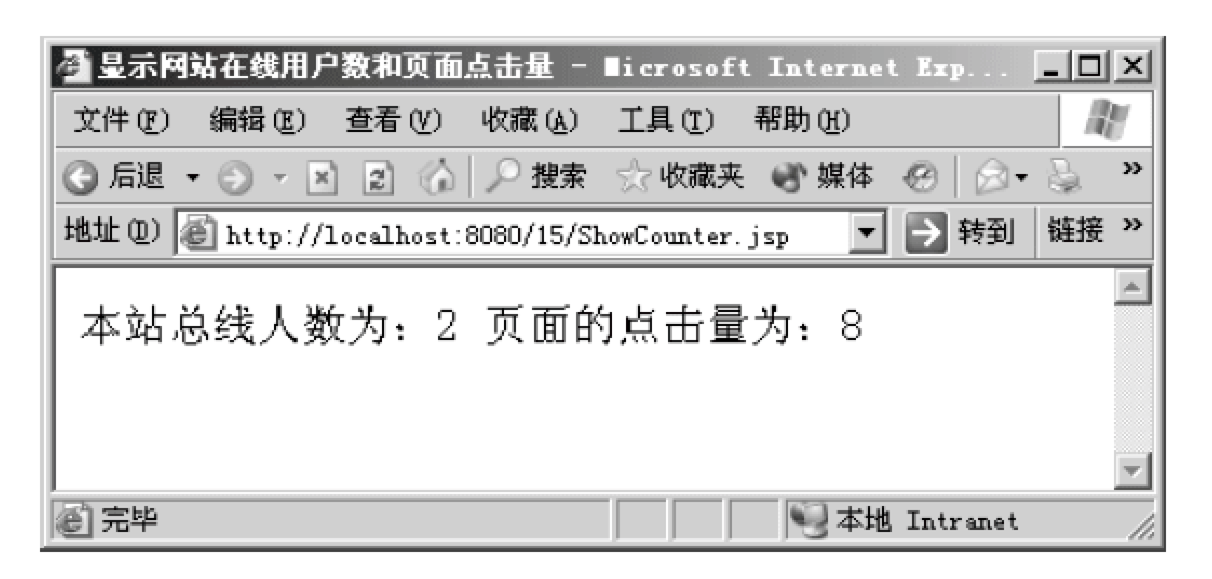


图 14.5　Servlet 监听器效果

14.7　小　　结

Servlet 监听器和过滤器是 Servlet 2.3 后新增加的一个重要特性，在很多应用中体现了很大的优势。在本章中介绍了 Servlet 监听器和过滤器的工作原理和使用方法，希望读者多多练习，更好地理解它们的工作原理。

第 15 章　JSP Web 应用的安全性

计算机网络的安全问题是一个让很多网络管理员头疼的问题，作为网络中的服务器，它暴露给很多用户，同时也增加了被入侵的风险。在提高应用的安全性方面，除了使用防火墙以及其他一些安全措施外，使用 JSP 技术提高 Web 应用的安全性也是很重要的方面，在本章中将介绍如何使用 JSP 技术及其相关的容器提高 Web 应用的安全性。

15.1　JSP/Servlet 容器认证

Servlet 规范中定义了一些组件用于实现容器认证的安全性，它是通过 web.xml 文件中的<security-constraint>元素、<login-config>元素和<security-role>元素体现的，下面分别介绍这 3 个元素。

1．<security-constraint>元素

<security-constraint>元素用来指定受保护的 Web 资源以及所有可以访问该 Web 资源的角色。例如在 Tomcat 的 Manager 应用中定义了如下的<security-constraint>元素：

```
<!----对这个 Web 应用定义了一个安全限制 -->
<security-constraint>
<!--  指定受包含的资源 --->
  <web-resource-collection>
    <web-resource-name>HTMLManger and Manager command</web-resource-name>
    <url-pattern>/jmxproxy/*</url-pattern>
    <url-pattern>/html/*</url-pattern>
    <url-pattern>/list</url-pattern>
    <url-pattern>/sessions</url-pattern>
    <url-pattern>/start</url-pattern>
    <url-pattern>/stop</url-pattern>
    <url-pattern>/install</url-pattern>
    <url-pattern>/remove</url-pattern>
    <url-pattern>/deploy</url-pattern>
    <url-pattern>/undeploy</url-pattern>
    <url-pattern>/reload</url-pattern>
    <url-pattern>/save</url-pattern>
    <url-pattern>/serverinfo</url-pattern>
    <url-pattern>/status/*</url-pattern>
    <url-pattern>/roles</url-pattern>
    <url-pattern>/resources</url-pattern>
```

```
    </web-resource-collection>

    <auth-constraint>
        <!-- 注意：这个角色在默认的用户文件中是不存在的 -->
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>
```

在这个<security-constraint>元素的定义中，<web-resource-collection>元素定义了所有受保护的 Web 资源，而<auth-constraint>子元素定义了可以访问这些保护的 Web 资源的角色为 Manager。

<security-constraint>元素还有一些其他的子元素，如表 15.1 所示。

表 15.1　security-constraint元素的子元素

子　元　素	描　　述
<web-resource-collection>	声明受保护的Web资源
<web-resource-name>	指定受保护资源的名字，<web-resource-collection>的子元素
<url-pattern>	受保护资源的URL路径，<web-resource-collection>的子元素
<http-method>	指定受保护的HTTP访问方法（包括DELETE、GET、POST、PUT），如果不定义，表示所有的方法都受保护，为<web-resource-collection>的子元素
<auth-constraint>	指定受保护资源的安全限制
<role-name>	指定可以访问受保护资源的角色名，为<auth-constraint>的子元素

2．<login-config>元素

<login-config>元素用来指定 Web 客户访问受保护的 Web 资源时，系统对客户进行验证的类型（具体表现为弹出不同登录对话框）。其中，在 Manager 应用中定义了如下的<login-config>元素：

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Tomcat Manager Application</realm-name>
</login-config>
```

当访问 Manager 应用时会弹出一个对话框，如图 15.1 所示。



图 15.1　BASIC 验证方式

这就是使用 BASIC 的验证方式。下面先介绍<login-config>元素的子元素，然后介绍几

种不同的验证方式。

<login-config>元素的子元素如表 15.2 所示。

表 15.2　<login-config>元素的子元素

子　元　素	描　　述
<auth-method>	系统对客户进行验证的类型，有BASIC、DIGEST和FORM 3种类型
<realm-name>	设定使用的Realm名称
<form-login-config>	使用FORM验证时的配置信息
<form-login-page>	使用FORM验证时的登录页面
<form-error-page>	使用FORM验证时的错误处理页面

例如在 admin 应用中定义的<login-config>元素如下：

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Tomcat Server Configuration Form-Based Authentication Area</realm-name>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

在上面的介绍中，可以知道系统对客户进行验证的类型，有 BASIC、DIGEST 和 FORM 3 种类型，在后面将会分别对这 3 种类型作简单的介绍。

3．<security-role>元素

<security-role>元素用于声明 Web 应用引用的角色名字，其中，在 Manager 应用中定义了如下的<security-role>元素：

```
<security-role>
  <description>
    The role that is required to log in to the Manager Application
  </description>
  <role-name>manager</role-name>
</security-role>
```

<security-role>元素只有两个子元素，分别介绍如下：

- <description>：对当前的<security-role>元素进行描述。
- <role-name>：引用的角色名称，对于多个名称要使用多个<role-name>。

15.1.1　使用基本认证（BASIC）

当有客户访问使用 BASIC 验证方式保护 Web 资源时，Tomcat 通过 HTTP Basic Authentication 方式弹出一个对话框（如图 15.1 所示）要求用户输入用户名和密码。在这种验证方式中，密码都是以 64 位的编码方式在网络上传输的。

使用 Basic Authentication 通常被认为是不安全的，因为它没有强健的加密方法，除非在客户端和服务器端都使用 HTTPS 或者其他密码加密方式（比如在一个虚拟私人网络中）。若没有额外的加密方法，网络管理员将能够截获（或滥用）用户的密码。

15.1.2　使用摘要认证（DIGEST）

DIGEST 指出客户机应该利用加密 Digest Authentication 形式传输用户名和口令，这提供了比 BASIC 验证更高的防范网络截取的安全性。

如果把 Manager 应用中的验证方法由 BASIC 改为 DIGEST，它会弹出如图 15.2 所示的对话框。

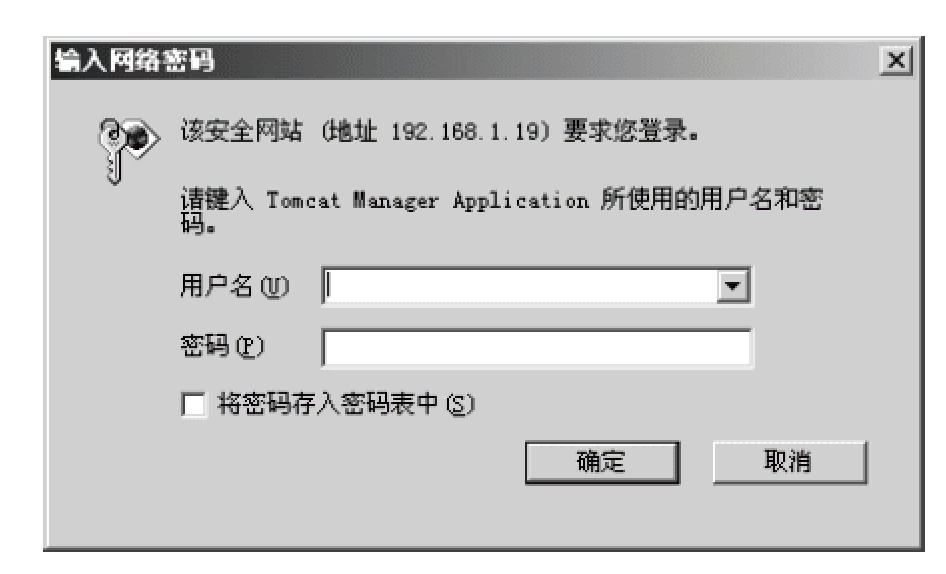


图 15.2　DIGEST 验证

15.1.3　使用基于表单的认证（FORM）

FORM 验证顾名思义，就是通过表单验证的方式（BASIC 和 DIGEST 验证是通过弹出标准对话框进行验证的）。

要定义一个 FORM 验证除了像其他的验证方式那样指定验证方式外，还需要指定登录页面和错误处理页面。用户提供的验证页面中必须提供一个登录表单，表单中的用户名和密码文本框的名字必须是 j_username 和 j_password，而且表单的 action 的值必须是 j_security_check。

下面介绍一个简单的验证页面和错误处理页面，进行表单验证，下面是登录页面 login.jsp 的源代码：

```
<%@ page contentType="text/html;charset=gb2312"%>
<html><head><title>FORM 验证</title></head>
<body >

<form action="j_security_check" method="post" >
<center>
<table>
  <tr>
    <td >Username:</td>
    <td ><input type="text" name="j_username" size="25"></td>
  </tr>
  <tr>
    <td >Password:</td>
    <td ><input type="password" name="j_password" size="25"></td>
  </tr>
</table>

</center>
  <center><br>
```

```
      <input type="submit" value="Login">
      <input type="reset" name="Reset" value="Reset">
    </center>
</form>

</body>
</html>
```

下面创建一个错误处理页面 errorpage.jsp，其源代码如下：

```
<%@ page contentType="text/html;charset=gb2312"%>
<html><head><title>Error Page</title></head>
<body>
        <p><font color=red size=+1>
                <b>用户名或密码错误</b>
        </font></p>
</body></html>
```

在 TomcatConfig 应用的 web.xml 文件中加入如下代码：

```
    <security-constraint>
        <display-name>Example Security Constraint</display-name>
        <web-resource-collection>
            <web-resource-name>Protected Area</web-resource-name>
        <!-- Define the context-relative URL(s) to be protected -->
        <!--    只有列出的方法是受保护的  -->
        <http-method>DELETE</http-method>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        <http-method>PUT</http-method>
        </web-resource-collection>
        <auth-constraint>
            <!-- Anyone with one of the listed roles may access this area -->
            <role-name>manager</role-name>
        </auth-constraint>
      </security-constraint>

      <!--    配置使用 FORM 的方式进行验证   -->
      <login-config>
        <auth-method>FORM</auth-method>
        <realm-name>Example Form-Based Authentication Area</realm-name>
        <form-login-config>
          <form-login-page>/login.jsp</form-login-page>
          <form-error-page>/errorpage.jsp</form-error-page>
        </form-login-config>
      </login-config>
```

这样就配置好了一个 FORM 验证，在浏览器中访问 TomcatConfig 应用的某个页面，它

会首先显示 login.jsp 页面，要求输入用户名和密码，效果如图 15.3 所示。

如果输入了错误的用户名和密码则会跳转到 errorpage.jsp 页面，如图 15.4 所示。



图 15.3　FORM 验证



图 15.4　错误处理页面

15.2　为 Web 应用配置使用 SSL

15.2.1　SSL 简介

SSL（Secure Socket Layer），安全套接字协议层是 Internet 上领先的安全协议机制，它是由 Netscape 公司开发的主要用于 Web 的安全传输协议。当 SSL 会话开始后，Web 浏览器将公共密钥发送给 Web 服务器，这样服务器可以很安全地将私人密钥发送给浏览器。浏览器和服务器在会话期间，用私人密钥加密数据。这种协议在 Web 上获得了广泛的应用。TLS（Transport Layer Security）、IETF（www.ietf.org）将 SSL 作了标准化，即 RFC2246，并将其称为 TLS。

采用 SSL 的 HTTP 称为 HTTPS 协议，其使用的默认端口是 443，而 HTTP 使用的默认协议是 80。下面介绍数字安全证书的概念和 SSL 的工作原理。

1．数字安全证书

数字安全证书是网络通信中标志通信各方身份信息的一系列数据，提供了一种在 Internet 上验证用户身份的方式，它一般是由一个由权威机构——CA 机构，又称为证书授权（Certificate Authority）中心发行的，人们可以在交往中用它来识别对方的身份。数字证书是一个经证书授权中心数字签名的包含公开密钥拥有者信息以及公开密钥的文件。最简单的证书包含一个公开密钥、名称以及证书授权中心的数字签名。一般情况下证书中还包括密钥的有效时间、发证机关（证书授权中心）的名称、该证书的序列号等信息，证书的格式遵循 ITUT X.509 国际标准。

2．SSL 工作原理

当客户浏览器与一个网站建立 HTTPS 连接时，用户的浏览器与 Web Server 之间要经过一个握手的过程来完成身份鉴定与密钥交换，从而建立安全连接。具体过程如下：

- 用户浏览器将其 SSL 版本号、加密设置参数、与 Session 有关的数据以及其他一些必要信息发送到服务器。

- ❑ 服务器将其 SSL 版本号、加密设置参数、与 Session 有关的数据以及其他一些必要信息发送给浏览器，同时发给浏览器的还有服务器的证书。如果配置服务器的 SSL 需要验证用户身份，还要发出请求要求浏览器提供用户证书。
- ❑ 客户端检查服务器证书，如果检查失败，提示不能建立 SSL 连接。如果成功，那么继续。
- ❑ 客户端浏览器为本次会话生成 pre-master secret，并将其用服务器公钥加密后发送给服务器。
- ❑ 如果服务器要求鉴别客户身份，客户端还要再对另外一些数据签名后并将其与客户端证书一起发送给服务器。
- ❑ 如果服务器要求鉴别客户身份，则检查签署客户证书的 CA 是否可信。如果不在信任列表中，结束本次会话。如果检查通过，服务器用自己的私钥解密收到的 pre-master secret，并用它通过某些算法生成本次会话的 master secret。
- ❑ 客户端与服务器均使用此 master secret 生成本次会话的会话密钥（对称密钥）。在双方 SSL 握手结束后传递任何消息均使用此会话密钥。这样做的主要原因是对称加密比非对称加密的运算量低一个数量级以上，能够显著提高双方会话时的运算速度。
- ❑ 客户端通知服务器此后发送的消息都使用这个会话密钥进行加密，并通知服务器客户端已经完成本次 SSL 握手。
- ❑ 服务器通知客户端此后发送的消息都使用这个会话密钥进行加密，并通知客户端服务器已经完成本次 SSL 握手。
- ❑ 本次握手过程结束，会话已经建立。双方使用同一个会话密钥分别对发送以及接收的信息进行加、解密。

15.2.2　在 Tomcat 中为 Web 应用配置使用 SSL

Tomcat 是一个优秀的 JSP/Servlet 容器，它可以作为单独的 Web 服务器，当然也可以在其上配置使用 SSL，配置 SSL 需要两个步骤：

（1）准备安全证书。

（2）配置 SSL 连接器。

下面就按照这两步在 Tomcat 上配置 SSL。

1．准备安全证书

在前面介绍到数字安全证书一般是由证书授权中心发行的，但事实上也可以自己制作自己的安全证书，不过它不具有权威性，也就是说，当客户看到安全证书上的信息时，并不能确定上面的信息和提供者的真实信息，但使用自制的安全证书可以保证网络上双方交换的信息不被窜改。下面介绍如何创建自己的安全证书。

Sun 提供了制作安全证书的工具 keytool。对于 JDK 1.4 以上版本可以在<JAVA_HOME>/bin 下找到这个工具，对于低版本的 JDK，也可以到 Sun 的网站下载这个工具：

```
http://java.sun.com/products/jsse/downloads/index.html
```

（1）打开 DOS 窗口（不需要修改运行目录），使用如下命令创建安全证书：

```
keytool –genkey –alias tomcat –keyalg RSA
```

其中各个选项的意义如下：

- –genkey：生成密钥对（如果没有）。
- –alias：指定密钥对的别名，该别名是公开的，这里指定为 Tomcat。
- –keyalg：指定加密算法，这里采用比较通用的算法 RSA。

（2）运行命令后可以看到提示信息，输入 keystore 的密码，这里输入 Tomcat 的默认密码 changit，默认的用户名就是命令中指定的 Tomcat。

（3）输入一些个人信息。

（4）在提示输入信息是否正确时，输入 y 表示信息正确。

（5）输入 Tomcat（上面 alias 的名字）的主密码，这里设置为与 keystore 相同就可以了（按回车键），效果如图 15.5 所示。

这样就可以在 Windows 的个人账户目录（C:\Documents and Settings\<UserName>）下找到一个.keystore 文件，这就是 keytool 生成的一个非对称密钥和自我签名的证书。

```
命令提示符
D:\Documents and Settings\zhw>keytool -genkey -alias tomcat -keyalg RSA
输入keystore密码：  changit
您的名字与姓氏是什么？
  [Unknown]：  sun
您的组织单位名称是什么？
  [Unknown]：  ms
您的组织名称是什么？
  [Unknown]：  ms
您所在的城市或区域名称是什么？
  [Unknown]：  bj
您所在的州或省份名称是什么？
  [Unknown]：  bj
该单位的两字母国家代码是什么
  [Unknown]：  cn
CN=sun, OU=ms, O=ms, L=bj, ST=bj, C=cn 正确吗？
  [否]：  y

输入<tomcat>的主密码
        （如果和 keystore 密码相同，按回车）：

D:\Documents and Settings\zhw>_
```

图 15.5　用 keytool 生成的安全证书

2．配置 SSL 连接器

在 Tomcat 的 Server.xml 中提供了配置 SSL 连接器的代码，只要把 Connector 的注释去掉就可以了，其代码如下：

```
<Connector port="8443"
           maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
           enableLookups="false" disableUploadTimeout="true"
           acceptCount="100" debug="0" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS" />
```

基于 SSL 的 HTTP（HTTPS）的默认端口为 443，这里将其改为 8443。Connector 的一些常用的配置属性的描述如表 15.3 所示。

表 15.3　Connector的一些常用的配置属性

属　性	描　述
clientAuth	如果想要Tomcat为了使用这个socket而要求所有SSL客户出示一个客户证书，置该值为true
keystoreFile	如果创建的keystore文件不在Tomcat认为的默认位置（一个在Tomcat运行的home目录下的叫做.keystore的文件），则加上该属性。可以指定一个绝对路径或依赖$CATALINA_BASE环境变量的相对路径
keystorePass	如果使用了一个与Tomcat预期不同的keystore（和证书）密码，则加入该属性
keystoreType	如果使用了一个PKCS12 keystore，加入该属性。有效值是JKS和PKCS12
sslProtocol	socket使用的加密/解密协议。如果使用的是Sun的JVM，则不建议改变这个值。据说IBM的1.4.1版的TLS协议的实现和一些流行的浏览器不兼容，这种情况下使用SSL
ciphers	此socket允许使用的被逗号分隔的密码列表。默认情况下，可以使用任何可用的密码
algorithm	使用的X509算法。默认为Sun的实现（SunX509）。对于IBM JVMS应该使用ibmX509。对于其他JVM，可以参考JVM文档取正确的值
truststoreFile	用来验证客户证书的TrustStore文件
truststorePass	访问TrustStore使用的密码。默认值是keystorePass
truststoreType	如果使用一个不同于正在使用的KeyStore的TrustStore格式，加入该属性。有效值是JKS和PKCS12

3．验证 SSL 配置的正确性

按照上述步骤配置好后，可以按照如下步骤验证 SSL 配置的正确性。

（1）启动 Tomcat，在浏览器地址栏中输入如下地址：https://localhost:8443，可以看到浏览器弹出安全警报窗口，如图 15.6 所示。

安全警报提示：您与该站点交换的信息不会被其他人查看或更改，但该站点的安全证书有问题。也就是说，该站点使用了安全证书，如果和这个网站通信，通信数据会被加密后在网络上传播是不会被其他人看到或修改的；另一方面，就如上面介绍的那样，这个网站的安全证书不是权威机构颁发的，不能确认对方的身份到底是不是与安全证书中的信息一致。也可以这样理解，该服务器向客户发送了一个“A”，使用了这个安全证书可以保证这个信息不会被更改也不会被其他人看到，但却无法保证这个信息“A”是否会对客户构成伤害。

（2）单击【查看证书】按钮，可以看到证书的版本、颁发者、有效期等相关信息，如图 15.7 所示。

这些信息都是使用 keytool 生成时输入的，可以看到这个安全证书的颁发者和接受证书的人是同一个人，不具有权威性。

（3）在安全警报对话框中单击【是】按钮，就会继续与 Tomcat 通信，浏览器显示页面如图 15.8 所示。

它和使用 HTTP 显示是一样的。

（4）在【安全警报】对话框中单击【否】按钮，将结束与 Tomcat 的通信。显示错误页面。

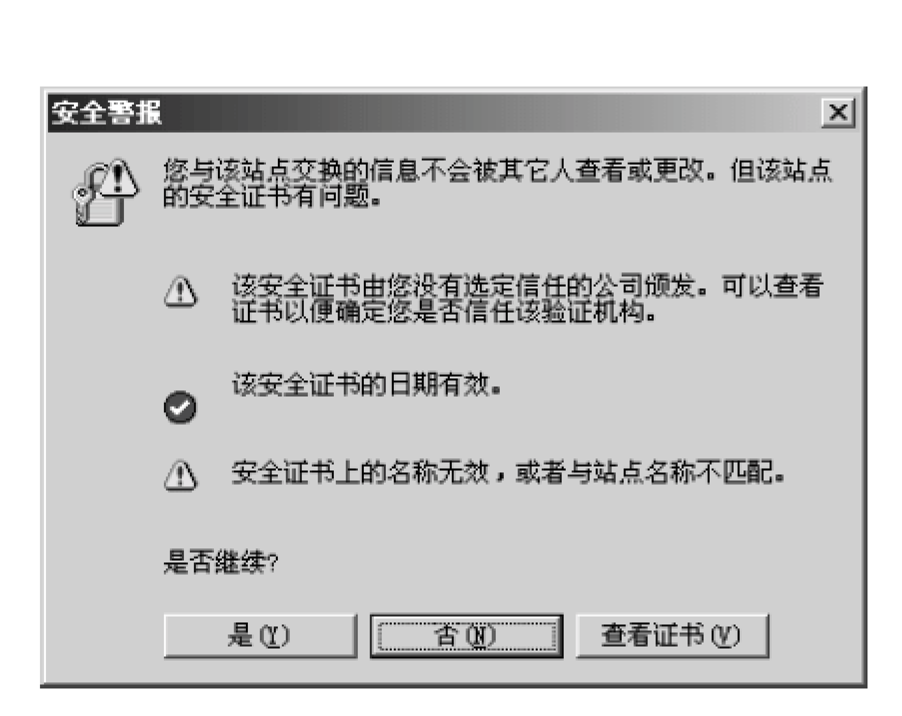


图 15.6　浏览器安全警报



图 15.7　安全证书详细信息

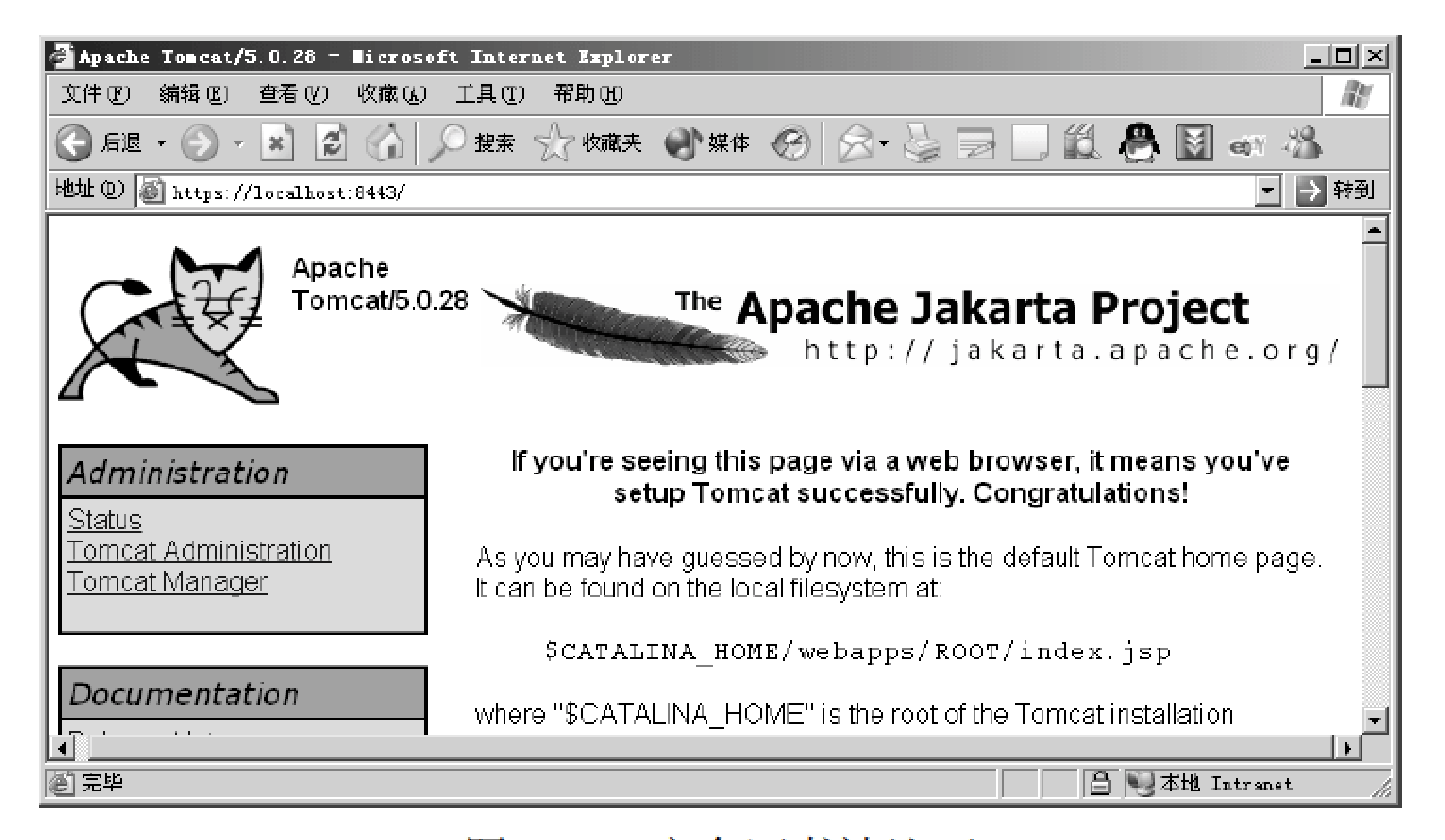


图 15.8　安全证书被认可

15.3　小　　结

JSP 技术在 Web 应用的开发中得到了广泛的应用，同时 Web 应用安全的问题也在挑战 JSP 技术，而 JSP 技术依托 Java 技术的良好安全性，所开发的 Web 应用也可以具有较好的安全性，但不同的 Web 服务器和 JSP 容器所提供的安全机制有所不同，在具体使用某个服务器时可以参考其相关文档。

第 3 部分

DESIGN

整合应用

第 16 章　Tomcat 容器的 JSP 特色应用

Tomcat 是 Servlet 规范的标准实现，同时，它也提供了很多有特色的技术，如 Tomcat 阀、安全域以及 Tomcat 实现的 JNDI 等。本章将对这些技术作简单介绍。

16.1　使用 Tomcat 阀

Tomcat 阀可以对 HTTP 请求进行预处理完成一些监测和管理功能，它的工作过程类似于 Servlet 过滤器，是 Tomcat 专有的特性，其他 Servlet 和 JSP 容器不支持此特性。下面介绍各种 Tomcat 阀的功能和使用方法。

所有的 Tomcat 阀都实现了 org.apache.catalina.Valve 接口或者继承了 org.apache.catalina.valves.ValveBase 类。Tomcat 阀可以分为 4 类：

- 客户访问日志阀。
- 远程地址过滤器阀。
- 远程主机过滤器阀。
- 客户请求记录阀。

它们可以被加入到 Engine、Host 和 Context 元素中。处在不同的元素中，阀的作用范围是不一样的。加入 Engine 元素的 Tomcat 阀可以预处理该 Engine 接收到的所有 HTTP 请求；加入到 Host 中的 Tomcat 阀可以预处理该 Host 接收到的所有 HTTP 请求。

配置 Tomcat 阀是很简单的，只需要在 Tomcat 配置文件 server.xml 中加入<Value>元素就可以了。具体形式为：

```
< Valve   className="..." ...... />
```

其中，就像 className 一样，每个 Tomcat 阀都有很多属性可以设置，不同的阀有不同的属性。下面就一一介绍各种 Tomcat 阀的相关属性和使用方法。

16.1.1　客户访问日志阀

客户访问日志阀（Access Log Valve）能够把客户的请求信息保存到日志文件中，这与标准服务器创建的日志格式是一样的。这些日志可以被标准日志工具分析，并对客户点击、用户活动会话等进行统计。它的很多配置和特性跟 File Logger 是相同的，例如每晚在零点时自动回滚等。客户访问日志阀可以和 Catalina 的任何容器关联，可以记下容器处理的所有请求。客户访问日志阀支持的配置属性如表 16.1 所示。

表 16.1　客户访问日志阀支持的配置属性

属　　性	描　　述
className	指定阀的Java实现类，这里应该为org.apache.catalina.valves.AccessLogValve
directory	日志存放的目录，可以是相对路径也可以是绝对路径，指定相对路径时是相对<TOMCAT_HOME>，默认值是logs
pattern	日志的格式和内容，默认是common
prefix	日志的前缀名，被加在日志文件的开头，默认是access_log，如果不要前缀，要指定一个长度为0的字符串
resolveHosts	如果设为true，把IP地址转化为主机名存到日志，否则直接记录IP地址
suffix	日志名的扩展名，默认是“”
rotatable	决定日志文件是否可旋转，默认为true
condition	条件日志，如果设置，只有在ServletRequest.getAttribute()为空时记录日志
fileDateFormat	自定义日志文件名中的日期格式

pattern 属性由字符串和带有“%”的 pattern 标志符前缀结合决定当前请求和响应信息的格式和内容。它支持如下的 pattern 码：

- %a：远程主机地址。
- %A：本地 IP 地址。
- %b：发送的字节数，不包括 HTTP Header，如果为零记录为“－”。
- %B：发送的字节数，不包括 HTTP Header。
- %h：远程主机名（如果属性 resolveHosts 设为 false 就是 IP 地址）。
- %H：请求使用的协议。
- %l：远程逻辑用户名（总是返回'-'）。
- %m：请求使用的方法（GET、POST 等）。
- %p：接受这个请求的本地端口。
- %q：请求中的查询字符串（如果存在以？开头）。
- %r：这个请求的第一行（请求的方法和请求的 URI）。
- %s：HTTP 响应的状态码。
- %S：用户会话 ID。
- %t：日期和时间，使用 Common Log 的格式。
- %u：经过安全认证的远程用户名，不存在时为'-'。
- %U：请求的 URL 路径。
- %v：本地服务器名。
- %D：处理请求花费的时间（以毫秒为单位）。
- %T：处理请求花费的时间（以秒为单位）。

它也支持记录 ServletRequest 中的一些信息，例如 Cookie、请求头和 Session 等。pattern 属性的默认值是 common，common 与 %h %l %u %t "%r" %s %b 是等同的。

在 Tomcat 的 server.xml 中的<Host>元素中加入如下元素：

```
<Valve className="org.apache.catalina.valves.AccessLogValve"
```

```
                directory="logs"   prefix="localhost_access_log." suffix=".txt" pattern= "common"
resolveHosts="false"/>
```

pattern 使用了默认值 common，重启 Tomcat 后，在浏览器地址栏中输入如下地址：http://localhost:8080/admin 并登录，然后查看 logs 目录下生成一个 localhost_access_log.2005-08-14.txt 文件。文件内容如下：

```
127.0.0.1 - - [14/Aug/2005:15:29:33 +0800] "GET /admin/ HTTP/1.1" 200 2578
127.0.0.1 - - [14/Aug/2005:15:29:39 +0800] "POST /admin/j_security_check HTTP/1.1" 302 -
127.0.0.1 - admin [14/Aug/2005:15:29:39 +0800] "GET /admin/ HTTP/1.1" 302 -
127.0.0.1 - admin [14/Aug/2005:15:29:39 +0800] "GET /admin/frameset.jsp HTTP/1.1" 200 946
127.0.0.1 - admin [14/Aug/2005:15:29:39 +0800] "GET /admin/blank.jsp HTTP/1.1" 200 579
127.0.0.1 - admin [14/Aug/2005:15:29:39 +0800] "GET /admin/banner.jsp HTTP/1.1" 200 1996
127.0.0.1 - admin [14/Aug/2005:15:29:41 +0800] "GET /admin/setUpTree.do HTTP/1.1" 200 8060
```

在这里使用了 pattern 的默认值 common，通过分析最后一条日志记录，看看它是如何和%h %l %u %t "%r" %s %b 对应起来的。

- %h：远程主机名为 127.0.0.1。
- %l：远程逻辑用户名为-。
- %u：经过安全认证的远程用户名为 admin。
- %t：日期和时间是[14/Aug/2005:15:29:41 +0800]。
- "%r"：客户请求的第一行，放在引号内为"GET /admin/setUpTree.do HTTP/1.1"。
- %s：服务器响应代码为 200。
- %b：发送的字节数为 8060。

16.1.2　远程地址过滤器

远程地址过滤器（Remote Address Filter）允许把客户的 IP 地址同一个或多个正则表达式相比较，从而决定是否为该客户提供服务。远程地址过滤器可以和 Catalina 的任何容器关联，而且在请求被处理之前它必须接受任何请求。

正则表达式与普通的通配符匹配是不相同的。具体关于正则表达式的相关问题可以参考相关文档。远程地址过滤器支持的配置属性介绍如表 16.2 所示。

表 16.2　远程地址过滤器支持的配置属性

属　性	描　述
className	指定阀的Java实现类，这里应该为org.apache.catalina.valves.RemoteAddrValve
allow	允许访问的客户IP地址正则表达式列表，用逗号分隔。如果这个属性被设置，只有匹配的地址才允许访问；如果不被设置，只要该地址不在拒绝列表中就允许访问
deny	拒绝访问的客户IP地址正则表达式列表，用逗号分隔

在 server.xml 中的<Host>元素中加入如下元素：

```
<Valve className="org.apache.catalina.valves.AccessLogValve"
        allow="127.0.0.*" deny ="222.*"/>
```

这样设置并重新启动 Tomcat 后，以 222 开头的 IP 地址将不被允许访问，并且只有 127.0.0 开头的 IP 地址才允许访问，也就是自己能够访问。

16.1.3　远程主机过滤器

远程主机过滤器（Remote Host Filter）和远程地址过滤器很相似。只不过远程主机过滤器根据远程客户的主机名来决定是否响应客户的请求，而不是客户的 IP 地址。在远程主机过滤器中，事先保存了一个允许访问的客户主机名列表和一个拒绝服务的客户主机名列表。如果客户的主机名在拒绝列表中，则这个客户的请求不会被 Tomcat 响应；如果客户的主机名在允许访问的列表中，那么这个客户的请求就会被 Tomcat 响应。远程主机过滤器支持的配置属性如表 16.3 所示。

表 16.3　远程主机过滤器支持的配置属性

属　　性	描　　述
className	指定阀的Java实现类，这里应该为org.apache.catalina.valves.RemoteHostValve
allow	允许访问的客户主机名正则表达式列表，用逗号分隔。如果这个属性被设置，只有匹配的地址才允许访问；如果不被设置，只要该地址不在拒绝列表中就允许访问
deny	拒绝访问的客户IP地址正则表达式列表，用逗号分隔

在 server.xml 中的<Host>元素中加入如下元素：

```
<Valve className="org.apache.catalina.valves.AccessLogValve"
        allow="ms*"/>
```

这样设置并重新启动 Tomcat 后，只有以 ms 开头的主机名才允许访问。

16.1.4　客户请求记录器

客户请求记录器（Request Dumper）是一个非常有用的工具，它可以调试与客户应用程序的交互。被配置后，会把和它相关的客户请求信息的详细细节保存到日志文件中。它使用的日志文件是在<Logger>中配置的。

注意：使用客户请求记录器是有副作用的，这个阀的输出包括请求中的任何参数。这些参数将会按照默认平台的编码标准解码。在应用程序中的后续任何对方法 request.setCharacterEncoding()的调用都不起作用

（1）客户请求记录器只有一个属性需要设置——lassName，它必须被设置为 org.apache.catalina.valves.RequestDumperValve。

（2）要使用客户请求记录器，必须保证在 Tomcat 的配置文件 server.xml 中已经配置了一个<Logger>元素。如果没有配置，可以按照如下配置：

```
<Logger className="org.apache.catalina.logger.FileLogger"
                directory="logs"    prefix="localhost_log." suffix=".txt"
          timestamp="true"/>
```

（3）配置好<Logger>元素后，在<Host>元素中加入如下元素：

```
<Valve className="org.apache.catalina.valves.RequestDumperValve"/>
```

（4）重新启动 Tomcat，在浏览器地址栏中输入地址 http://localhost:8080/admin/后，查看刚刚配置好的日志文件。可以看到文件内容如下：

```
2005-08-14 16:40:37 RequestDumperValve[Catalina]: REQUEST URI          =/admin/
2005-08-14 16:40:37 RequestDumperValve[Catalina]:              authType=null
2005-08-14 16:40:37 RequestDumperValve[Catalina]:   characterEncoding=null
2005-08-14 16:40:37 RequestDumperValve[Catalina]:        contentLength=-1
2005-08-14 16:40:37 RequestDumperValve[Catalina]:          contentType=null
2005-08-14 16:40:37 RequestDumperValve[Catalina]:          contextPath=/admin
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
contextPath =/admin
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
cookie=JSESSIONID=835C7E753B5293F4D9CA31DAEE2269CE
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=accept=image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, application/x-shockwave-flash, */*
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=accept-language=zh-cn
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=accept-encoding=gzip, deflate
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=user-agent=Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=host=localhost:8080
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=connection=Keep-Alive
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=cookie=JSESSIONID=835C7E753B5293F4D9CA31DAEE2269CE
2005-08-14 16:40:37 RequestDumperValve[Catalina]:                locale=zh_CN
2005-08-14 16:40:37 RequestDumperValve[Catalina]:                method=GET
2005-08-14 16:40:37 RequestDumperValve[Catalina]:              pathInfo=null
2005-08-14 16:40:37 RequestDumperValve[Catalina]:              protocol=HTTP/1.1
2005-08-14 16:40:37 RequestDumperValve[Catalina]:          queryString=null
2005-08-14 16:40:37 RequestDumperValve[Catalina]:           remoteAddr=127.0.0.1
2005-08-14 16:40:37 RequestDumperValve[Catalina]:           remoteHost=127.0.0.1
2005-08-14 16:40:37 RequestDumperValve[Catalina]:           remoteUser=null
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
requestedSessionId=835C7E753B5293F4D9CA31DAEE2269CE
2005-08-14 16:40:37 RequestDumperValve[Catalina]:                scheme=http
2005-08-14 16:40:37 RequestDumperValve[Catalina]:           serverName=localhost
2005-08-14 16:40:37 RequestDumperValve[Catalina]:           serverPort=8080
2005-08-14 16:40:37 RequestDumperValve[Catalina]:          servletPath=/index.jsp
2005-08-14 16:40:37 RequestDumperValve[Catalina]:              isSecure=false
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
---------------------------------------------------------------
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
---------------------------------------------------------------
2005-08-14 16:40:37 RequestDumperValve[Catalina]:              authType=null
```

```
2005-08-14 16:40:37 RequestDumperValve[Catalina]:         contentLength=-1
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
contentType=text/html;charset=utf-8
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
cookie=JSESSIONID=8EA671C9AF6E6B522139653A4180F6F5; domain=null; path=/admin
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=Pragma=No-cache
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=Cache-Control=no-cache
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
header=Expires=Thu, 01 Jan 1970 08:00:00 CST
2005-08-14 16:40:37 RequestDumperValve[Catalina]: header=Set-Cookie=JSESSIONID=8EA671
C9AF6E6B522139653A4180F6F5; Path=/admin
2005-08-14 16:40:37 RequestDumperValve[Catalina]:              message=null
2005-08-14 16:40:37 RequestDumperValve[Catalina]:           remoteUser=null
2005-08-14 16:40:37 RequestDumperValve[Catalina]:               status=200
2005-08-14 16:40:37 RequestDumperValve[Catalina]:
===============================================================
```

16.1.5　单点登录阀

当开发者希望用户只要一登录与本虚拟主机相关的 Web 应用程序，用户就可以被同一台虚拟主机上的所有 Web 应用程序识别时，可以使用单点登录阀（Single Sign On Valve）。单点登录阀的配置属性如表 16.4 所示。

表 16.4　单点登录阀支持的配置属性

属　　性	描　　述
className	指定阀的Java实现类，这里必须为org.apache.catalina.authenticator.SingleSignOn
debug	这个组件创造的调试信息的级别，默认为0，也就是没有任何输出
requireReauthentication	决定每个请求需要到安全Realm中进行验证，如果为true，则这个阀用缓存的信任信息重新认证。默认为false

在 server.xml 中配置一个单点登录阀元素如下：

```
<Valve className="org.apache.catalina.authenticator.SingleSignOn"
                      debug="0"/>
```

这样就可以实现单点登录的功能了。不过使用单点登录还有一些限制。如下：

- Valve 必须被配置和嵌套在相同的 Host 元素中，并且所有需要进行单点验证的 Web 应用（必须通过 Context 元素定义）都位于该 Host 下。
- 包括共享用户信息的 Realm 必须被设置在同一级 Host 中或者嵌套之外。
- 不能被 Context 中的 Realm 覆盖。
- 使用单点登录的 Web 应用最好使用一个 Tomcat 内置的验证方式（被定义在 web.xml 中），这比自定义的验证方式好。Tomcat 内置的验证方式包括 BASIC、DIGEST、FORM 和 CLIENT-CERT。
- 如果使用单点登录，还希望集成一个第三方的 Web 应用到应用中来，并且这个新

的 Web 应用使用它自己的验证方式，而不使用容器管理安全，那请求第三方应用时还需要重新登录。单点登录需要使用 Cookie。

16.2　使用基于 JNDI 的应用程序开发（介绍 Tomcat 的 JNDI 资源）

JNDI 是 The Java Naming and Directory Interface 的简写。使用 JNDI 可以访问命名和目录服务。实际是一组在 Java 应用中访问命名和目录服务的 API。命名服务将名称和对象联系起来，使得开发者可以用名称访问对象。

Tomcat 5 为每一个运行其中的 Web 应用程序提供了一个 JNDI InitialContext 的实例，它提供服务的方式和 J2EE 服务器提供服务的方式是一样的。J2EE 标准在 Web 应用的 /WEB-INF/web.xml 文件中提供了一些标准元素用于引用资源，被引用的资源必须在服务器应用的配置文件中配置指定。

对于 Tomcat 5 来说，这些资源必须配置在<Context>元素或者<Server>元素下的<DefaultContext>元素中。

如果配置在<Context>元素中，这个资源只可以在具体某个应用的 web.xml 文件中设置。如果配置在<DefaultContext>元素中，就必须修改<TOMCAT_HOME>\conf\server.xml 文件。

Tomcat 5 为整个服务器保留了一个全局资源的命名空间，它们可以在<TOMCAT_HOME>\conf\server.xml 文件中配置，然后使用<ResourceLink>让各个应用程序可以访问这个资源。

在这些元素中定义的资源可以被具体某个应用的配置文件/WEB-INF/web.xml 中的某些元素引用。这些元素可以是如下几项：

- <env-entry>：环境入口，设置应用程序如何操作。
- <resource-ref>：资源参数，一般是数据库驱动程序、Java Mail Sessions、自定义类工厂等。
- <resource-env-ref>：在 Servlet 2.4 中用来简化设置不需认证信息的资源，如环境参数、resource-ref 变量。

InitialContext 在应用程序初始化发布时被配置好，之后应用程序可以一直使用这个对象。所有资源的入口都在 java:comp/env 这个 JNDI 命名空间下。对于 JDBC DataSource 来说，可以使用如下代码：

```
// 获得环境的命名上下文
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");

//寻找需要的 DataSource
DataSource ds = (DataSource)
  envCtx.lookup("jdbc/EmployeeDB");

// 从连接池中定位并获取一个连接
```

```
Connection conn = ds.getConnection();
使用这个连接做访问数据库的操作
conn.close();
```

任何可用 JNDI 资源都需要使用在<Context>或者<DefaultContext>下的元素进行配置。这些元素如下：

- <Environment>：设置一个可变的 JNDI InitialContext 入口的名字和值（同上面所说的<env-entry>等价）。
- <Resource>：设置应用程序可用资源的名字和类型（同上面所说的<resource-ref>等价）。
- <ResourceParams>：设置 Java 资源类工厂的名称或将用的 JavaBeans 属性。
- <ResourceLink>：给全局 JNDI 环境（JNDI Context）添加一个链接。

Tomcat 包含了一些标准资源工厂以用于为 Web 应用程序提供服务。要使用这些服务是需要修改一些配置文件（像上面介绍的那些一样）。除了使用 Tomcat 提供的资源工厂外，也可以自己设计资源工厂。下面就介绍如何使用 Tomcat 的标准资源工厂以及如何设计安装自己定制的资源工厂。

16.2.1　使用通用 JavaBeans 资源

这个资源工厂可以创建任何符合标准 JavaBeans 命名约定的 Java 类对象（例如没有参数的构造器、每个属性都有对应的 setter 和 getter 方法），每次使用 lookup()方法查找一个对象时，这个资源工厂都会创建一个 Bean 实例。

下面就介绍一个简单使用普通 JavaBeans 资源的例子。在这个例子中，创建了一个保存用户信息的 JavaBeans，然后从 JSP 页面中访问使用这个 Bean。

1．创建 JavaBeans 类

JavaBeans 资源每次被搜索时，JavaBeans 类都会被实例化一次。下面创建了一个 cn.ac.ict.UserInfo 的 JavaBeans。如下：

```
package cn.ac.ict;

public class UserInfo{
private String username ="guest";
private String password ="anonymous";
//下面是 username 属性的设置和获取方法
public void setUsername (String user){
	username = user;
}

public String getUsername(){
	return this.username;
}
//下面是 password 属性的设置和获取方法
public void setPassword(String pass){
	this.password = pass;
```

```
}

public String getPassword(){
        return this.password;
}

}
```

这个 JavaBeans 是非常简单的，有两个属性 username 和 password 以及相应的 setter 方法和 getter 方法。

2．配置 Tomcat 资源工厂

要配置 Tomcat 资源工厂，就需要修改<TOMCAT_HOME>/conf/server.xml 文件或者<TOMCAT_HOME>/conf/Catalina/localhost/目录下对应 Web 应用的 Context 名字的 XML 文件。

资源的声明可以被放在<Context>、<Host>或<Engine>元素下。如果放在<Context>下，这个资源工厂可以被这个应用使用；而放在<Host>或<Engine>元素下，就使得这个资源工厂有了更大的有效范围，分别在本 Host 和本 Engine 中有效。

如果修改<TOMCAT_HOME>/conf/Catalina/localhost/目录下对应 Web 应用的 Context 名称的 XML 文件，则只能使这个资源工厂在本 Web 应用范围内有效。

下面以<TOMCAT_HOME>/conf/Catalina/localhost/目录下对应 Web 应用的 Context 名称的 XML 文件为例添加资源工厂，假定本 Web 应用的 Context 名为 JNDI，则在<TOMCAT_HOME>/conf/Catalina/localhost/JNDI.xml 文件的<Context>元素中添加如下语句：

```
  <!---定义了一个资源-->
<Resource name="bean/userinfo" auth="Container"
                type="cn.ac.ict.UserInfo"/>

<!---下面为名为"bean/userinfo"的资源指定需要使用的参数--->
  <ResourceParams name="bean/userinfo">
    <parameter>
      <name>factory</name>
      <value>org.apache.naming.factory.BeanFactory</value>
    </parameter>
    <parameter>
      <name>username</name>
      <value>rambler</value>
    </parameter>
  </ResourceParams>
```

在上面的语句中首先使用<Resource>定义了一个资源，名称为 bean/userinfo，由 Container 授权，类型为 cn.ac.ict.UserInfo。下面的<ResourceParams>为名为 bean/userinfo 的资源指定了参数，其中 factory 是一个必需的参数，用于指定实例化这个 Bean 所需要的工厂。其他的参数可以根据具体 Bean 的不同来设置。这里设置了 username 属性的值为 rambler。

注意： 这个资源的名字 bean/userinfo 是应用程序引用资源时使用的惟一性标志。

3．声明资源引用

为了在某个 Web 应用中使用在容器组件中定义的（例如上面的资源定义在 Context 组

件中）资源，还需要在 Web 应用的 web.xml 文件中声明引用的资源。修改 JNDI 应用下的/WEB-INF/web.xml 文件，在<Context>元素中添加如下语句：

```
<resource-env-ref>
  <description>
    Object factory for UserInfo instances.
  </description>
<!---    指定了资源的名字   --->
  <resource-env-ref-name>
    bean/userinfo
  </resource-env-ref-name>
<!---    指定了资源的类型   --->
  <resource-env-ref-type>
    cn.ac.ict.UserInfo
  </resource-env-ref-type>
</resource-env-ref>
```

其中，<resource-env-ref-name>元素确定了使用的资源名称（在 TomcatConfig.xml 文件中定义的 bean/userinfo），然后使用元素<resource-env-ref-type>指定了资源的类型为 cn.ac.ict.UserInfo。

4．使用 JavaBeans 资源

经过上面的准备工作，需要使用的 JavaBeans 资源已经被配置好了，这样就可以在 JSP 页面中访问这个 JavaBeans 资源了。下面是一个测试使用这个 JavaBeans 资源的一个 JSP 文件的源代码（命名为 javabeansource.jsp）：

```
<%@ page import="javax.naming.*,cn.ac.ict.*"%>
<html>
    <head><title>Generic JavaBean Resources Test</title></head>
<body>
<h2>Generic JavaBean Resources</h2>

<%
//获取上下文对象
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
//寻找需要实例化的资源对象
UserInfo bean = (UserInfo) envCtx.lookup("bean/userinfo");
//输出资源的信息
out.println("Username = " + bean.getUsername() + ", password = " +
bean.getPassword());

%>
</body>
</html>
```

上述代码通过 Context initCtx = new InitialContext();获得了一个关于本 JSP 页面应用的上下文（Context）对象 initCtx，然后通过这个对象查找已定义好的资源 bean/userinfo，获得一个 JavaBeans 资源。

可以看到，JSP 代码的输出部分输出了 Bean 中保存的用户名和密码。在浏览器地址栏

中输入地址 http://localhost:8080/JNDI/javabeansource.jsp 便可看到页面显示如图 16.1 所示。

其中的用户名显示为 rambler，也就是在配置资源工厂时使用<ResourceParams>元素设定的，而密码则没有重设值，所以输出默认值 anonymous。

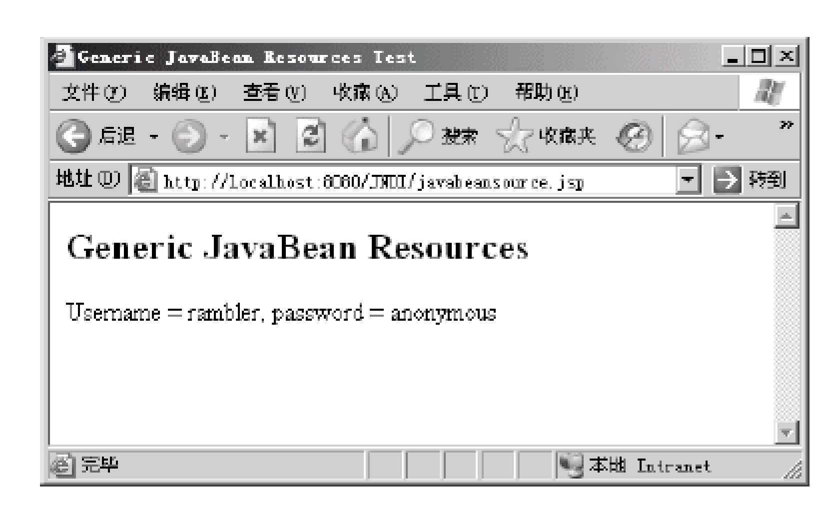


图 16.1　访问 JavaBeans 资源

16.2.2　使用 Java Mail Sessions 资源

在很多 Web 应用中，发送电子邮件是系统功能要求的一部分。Java Mail API 使得这个过程简单化了，但还是需要很多的配置细节。如果写在程序中会使得程序变得复杂和不易维护。

Tomcat 5 中提供了一个标准资源工厂以用于创建 javax.mail.Session 的实例，而且这个实例是已经连接到 SMTP 服务器的。这样就可以使应用程序不必配置邮件服务器的细节，从而使维护变得容易了。

使用 Java Mail Sessions 的基本步骤如下：

（1）配置 Tomcat 资源工厂。

（2）声明资源引用。

（3）代码实现。

16.2.3　使用 JDBC Data Sources

许多 Web 应用程序都需要使用 JDBC 驱动程序连接数据库。在 J2EE 平台规范中，允许实现一个 JDBC Data Sources（也就是 JDBC 的数据库连接池），Tomcat 5 对这个实现提供了很好的支持，使得 Web 应用可以不需要修改就能移植到其他的平台上。

注意： Tomcat 5 默认对数据源的支持使用了 Jakarta Commons 子项目的 DBCP 数据库连接池，也可以使用其他的实现技术，只是需要实现 javax.sql.DataSource 就可以了。

在 Tomcat 5 中配置使用 JDBC Data Sources 可以按照如下步骤进行：

（1）安装 JDBC 驱动程序。

（2）配置 Tomcat 资源工厂。

（3）声明资源引用。

（4）代码实现。

16.3　小　　结

Tomcat 得到了广泛的认可，它不但完全实现了 Servlet/JSP 的最新标准，而且还提供了很多优良的特性。本章只介绍了有代表性的两个。读者在使用 Tomcat 时会发现还有很多独有的特性可以优化 Web 开发。当然其他的 Servlet 容器也有自己的特性，这里就不一一介绍了。读者可以参考 Tomcat 等 Servlet 容器的相关文档。

第 17 章　在 JSP 中使用 Hibernate 实现数据持久化

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，使得 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。Hibernate 可以应用在任何使用 JDBC 的场合，既可以在 Java 的客户端程序实用，也可以在 Servlet/JSP 的 Web 应用中使用，最具革命意义的是，Hibernate 可以在应用 EJB 的 J2EE 架构中取代 CMP，完成数据持久化的重任。

17.1　快速体验 JSP 结合 Hibernate——JSP 和 Hibernate 结合的简单例子

17.1.1　Hibernate 简介

Hibernate 是一个面向 Java 环境的对象/关系数据库映射工具。对象/关系数据库映射（Object/Relational Mapping，ORM）这个术语表示一种技术，用来把对象模型表示的对象映射到基于 SQL 的关系模型数据结构中去。

Hibernate 不仅管理 Java 类到数据库表的映射(包括 Java 数据类型到 SQL 数据类　型的映射)，还提供数据查询和获取数据的方法，可以大幅度减少开发时人工使用 SQL 和 JDBC 处理数据的时间。

Hibernate 可以帮助开发者消除或者包装那些针对特定厂商的 SQL 代码，并且帮助开发者把结果集从表格式的表示形式转换到一系列的对象去，Hibernate 使用数据库和配置信息为应用程序提供持久化服务（以及持久的对象）。Hibernate 的目标是对于开发者通常的数据持久化相关的编程任务解放其中的 95%。对于以数据为中心的程序来说，它们往往只在数据库中使用存储过程来实现商业逻辑，Hibernate 可能不是最好的解决方案；对于那些在基于 Java 的中间层应用中，实现面向对象的业务模型和商业逻辑的应用中 Hibernate 是最有用的。Hibernate 体系结构如图 17.1 所示。

Hibernate 本身是个独立的框架，它不需要任何 Web Server 或 Application Server 的支持。下面详细看一下 Hibernate 运行时体系结构。由于 Hibernate 非常灵活，且支持数种应用方案，所以这里只描述两种极端的情况。“轻型”的体系结构方案，要求应用程序提供自己的 JDBC 连接并管理自己的事务。这种方案使用了 Hibernate API 的最小子集，此时，Hibernate 在应

用程序中的作用如图 17.2 所示。

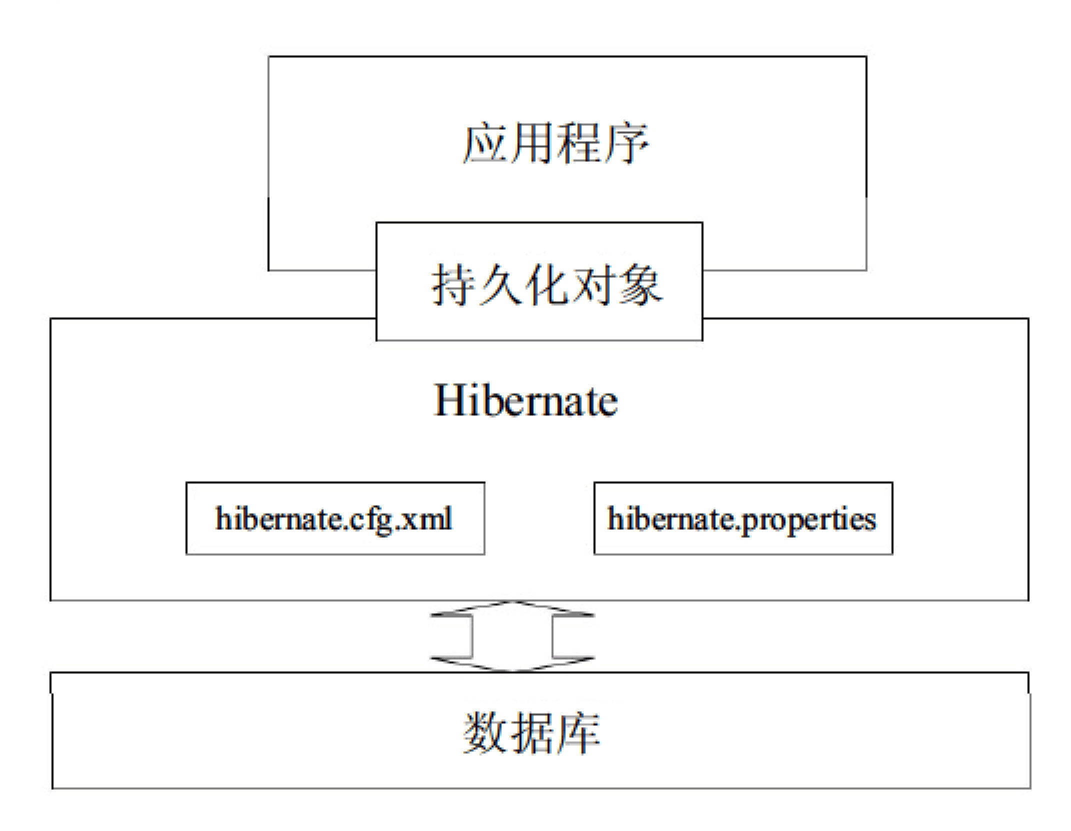


图 17.1　Hibernate 体系结构

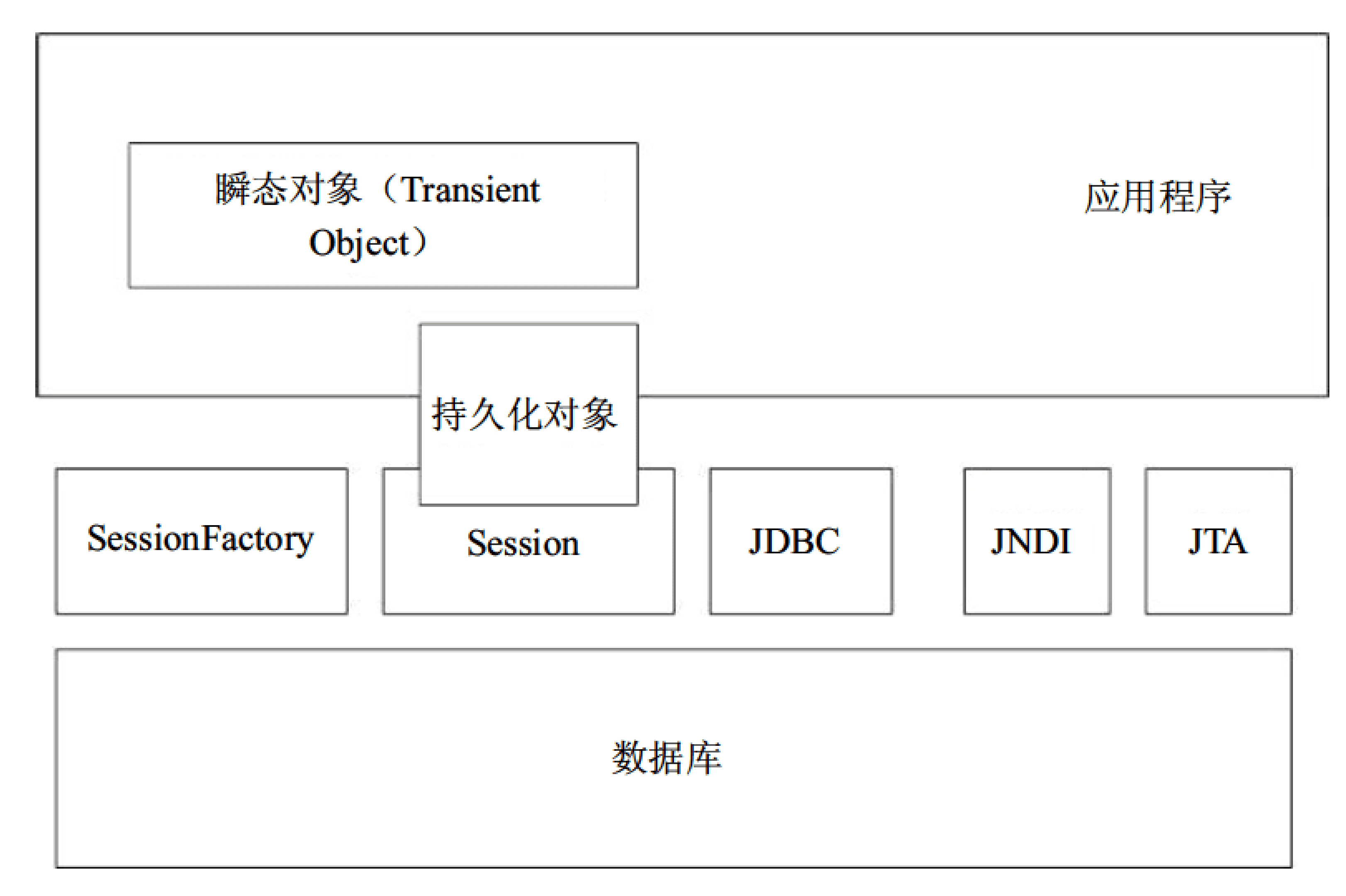


图 17.2　轻型 Hibernate 运行体系结构

“全面解决”的体系结构方案将应用层从底层的 JDBC/JTA API 中抽象出来，而让 Hibernate 来处理这些细节，如图 17.3 所示。

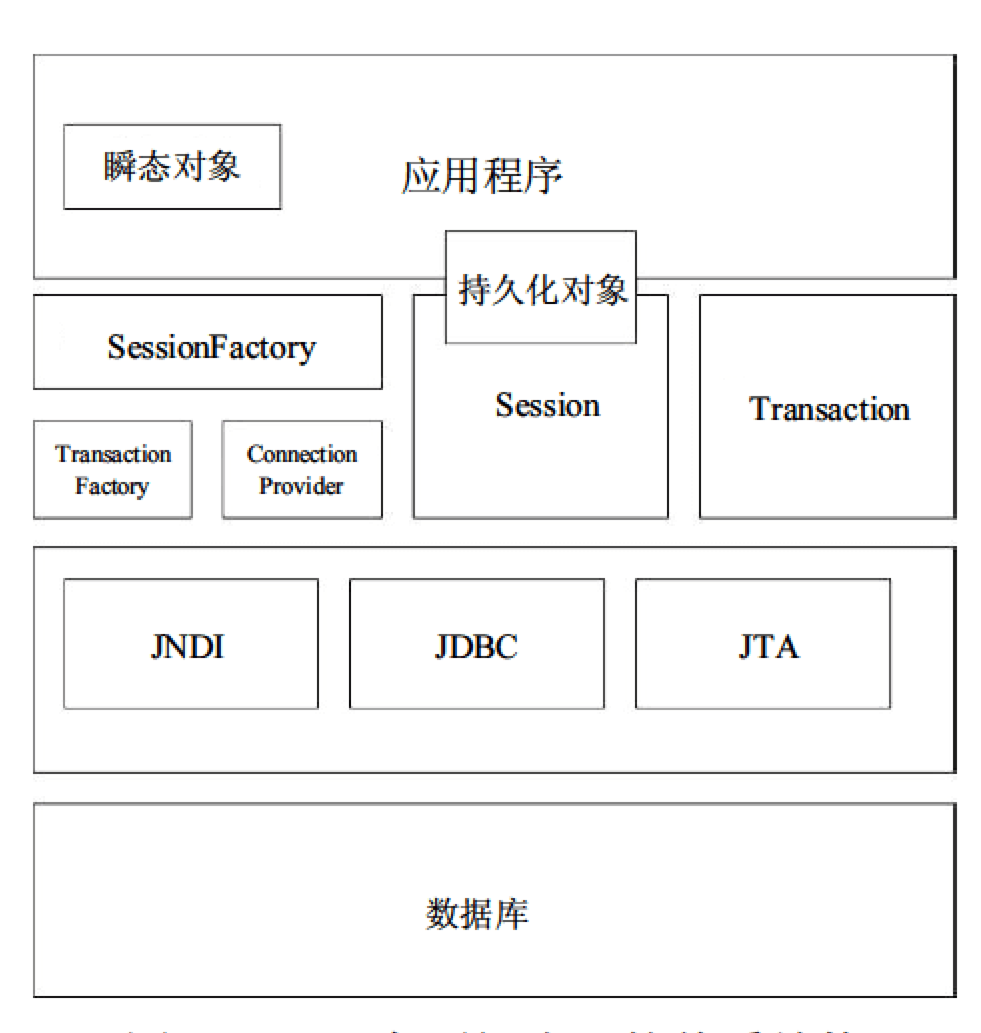


图 17.3　“全面解决”的体系结构

图 17.3 中各个对象的介绍如下：

- SessionFactory (org.hibernate. SessionFactory)：针对单个数据库映射关系经过编译后的内存镜像，它也是线程安全的（不可变）。它是生成 Session 的工厂，本身要用到 ConnectionProvider。该对象可以在进程或集群的级别上，为那些事务之间可以重用的数据提供可选的二级缓存。

- Session(org.hibernate.Session)：表示应用程序与持久储存层之间交互操作的一个单线程对象，此对象生存期很短。它隐藏了 JDBC 连接，也是 Transaction 的工厂。会持有一个针对持久化对象的必选（第一级）缓存，在遍历对象图或者根据持久化标识查找对象时会用到。
- 持久的对象及其集合：带有持久化状态的、具有业务功能的单线程对象，此对象生存期很短。这些对象可以是普通的 JavaBeans 或者 POJO（Plain Old Java Objects，就是 POJOs，有时也称作 Plain Ordinary Java Objects，一个 POJO 很像 JavaBeans），比较特殊的是它们只与一个（仅仅一个）Session 相关联。这个 Session 被关闭的同时，这些对象也会脱离持久化状态，可以自由地被应用程序的任何层使用（例如，用于跟表示层打交道的数据传输对象 Data Transfer Object）。
- 瞬态（Transient）以及脱管（Detached）的对象及其集合：持久类的没有与 Session 相关联的实例。它们可能是在被应用程序实例化后尚未进行持久化的对象。也可能是因为实例化它们的 Session 已经被关闭而脱离持久化的对象。
- 事务 Transaction(org.hibernate.Transaction)（可选的）：应用程序用来指定原子操作单元范围的对象，它是单线程的，生存期很短。它通过抽象将应用从底层具体的 JDBC、JTA 以及 CORBA 事务隔离开。某些情况下，一个 Session 之内可能包含多个 Transaction 对象。尽管是否使用该对象是可选的，但事务边界的开启与关闭（无论是使用底层的 API 还是使用 Transaction 对象）是必不可少的。
- ConnectionProvider(org.hibernate.connection.ConnectionProvider)（可选的）：生成 JDBC 连接的工厂（同时也起到连接池的作用）。它通过抽象将应用从底层的 Datasource 或 DriverManager 中隔离开。仅供开发者扩展/实现用，并不暴露给应用程序使用。
- TransactionFactory (org.hibernate.TransactionFactory)（可选的）：生成 Transaction 对象实例的工厂。仅供开发者扩展/实现用，并不暴露给应用程序使用。
- 扩展接口：Hibernate 提供了很多可选的扩展接口，可以通过实现它们来定制持久层的行为。

在一个“轻型”的体系结构中，应用程序可能绕过 Transaction/TransactionFactory 以及 ConnectionProvider 等 API 直接跟 JTA 或 JDBC 打交道。

17.1.2　配置 Hibernate 环境

下面以在 Apache Tomcat Servlet 容器中为 Web 应用程序配置使用 Hibernate 3.0（使用 Tomcat 5.0 版本）为例，介绍一个简单的 Hibernate 与 JSP 的结合应用。Hibernate 在大多数主流 J2EE 应用服务器的运行环境中都可以良好地工作，甚至也可以在独立 Java 应用程序中使用。使用的示例数据库系统是 MySQL 4.1，而且只需要修改 Hibernate SQL 语言配置与连接属性，就可以很容易地支持其他数据库了。

要使用 Hibernate 开发 Web 应用，首先要到其官方网站 http://www.hibernate.org/下载

Hibernate 包，在本实例中使用的是 Hibernate 3.0，把 hibernate-3.0.zip 解压缩后，可以在其解压目录下看到一个 hibernate3.jar 文件，这就是 Hibernate 运行需要的核心包，但只有这一个包是不够的，还需要 lib 目录下的其他包的支持，要使用 Hibernate 就需要把这些包一起复制到 Web 应用的 WEB-INF\lib 目录下，但这些包并不都是必需的，根据具体的需要可以有选择地使用，下面对一部分包的的作用作一下简单介绍：

- ❑ antlr-2.7.5H3.jar（必需）：Hibernate 使用 ANTLR 来产生查询分析器，这个类库在运行环境下时也是必需的。
- ❑ cglib-2.1.jar（必需）：Hibernate 用它来实现 PO 字节码的动态生成，非常核心的库必须使用的 jar 包。
- ❑ asm.jar（必需）：Hibernate 在运行时使用这个代码生成库增强类（与 Java 反射机制联合使用）。
- ❑ commons-collections-2.1.1.jar（必需）：Apache Commons 包中的一个，包含了一些 Apache 开发的集合类，功能比 java.util.*强大。必须使用的 jar 包。
- ❑ commons-logging-1.0.4.jar（必需）：Apache Commons 包中的一个，包含了日志功能，必须使用的 jar 包。这个包本身包含了一个 Simple Logger，但功能很弱。在运行时它会先在 CLASSPATH 中找 Log4j，如果有，就使用 Log4j，如果没有，就找 JDK1.4 带的 java.util.logging，如果还找不到就使用 Simple Logger。
- ❑ ehcache-1.1.jar（必需）：Hibernate 可以使用不同 cache 缓存工具作为二级缓存。EHCache 是默认的 cache 缓存工具。
- ❑ dom4j.jar（必需）：dom4j 是一个 Java 的 XML API，类似于 jdom，用来读写 XML 文件。dom4j 是一个非常优秀的 Java XML API，具有性能优异、功能强大和非常易于使用的特点，同时它也是一个开放源代码的软件，可以在 SourceForge 上免费下载。Sun 的 JAXM 也在用 dom4j。这是必须使用的 jar 包，Hibernate 用它来读写配置文件。
- ❑ Log4j.jar（可选）：Hibernate 使用 Commons Logging API，它也可以依次使用 Log4j 作为底层实施 Log 的机制。如果上下文类目录中存在 Log4j 库，则 Commons Logging 使用 Log4j 和它在上下文类路径中寻找的 log4j.properties 文件。可以使用在 Hibernate 发行包中包含的那个示例 Log4j 的配置文件。这样，把 Log4j.jar 和它的配置文件（位于 src/目录中）复制到程序的上下文类路径下，就可以在后台看究竟程序是如何运行的。

17.1.3　准备数据库和数据库连接池

在 MySQL 的数据库管理系统中建立一个数据库，名为 hibernateTest，并在这个数据库中建立一个数据表，名为 productlist，其中表的字段和描述如表 17.1 所示。

表 17.1　productlist表的描述

字　段　名	类　　型	描　　述
pid	varchar(15)	主键，非空
pname	varchar(30)	非空
pcomp	varchar(30)	非空
madew	varchar(20)	可为空
madedate	date	可为空
rprice	float	可为空
cprice	float	可为空
count	int(11)	可为空
description	varchar(100)	可为空

建立 productlist 表的 SQL 语句如下：

```
CREATE TABLE 'productlist' (
  'pid' varchar(15) NOT NULL default '',
  'pname' varchar(30) NOT NULL default '',
  'pcomp' varchar(30) NOT NULL default '',
  'madew' varchar(20) default NULL,
  'madedate' date default NULL,
  'rprice' float default NULL,
  'cprice' float default NULL,
  'count' int(11) default NULL,
  'description' varchar(100) default NULL,
  PRIMARY KEY   ('pid')
);
```

数据库建立好后，然后在 web.xml 文件中配置数据库连接池。下面是 web.xml 文件的完整内容，具体 Context 的路径等信息应根据具体情况修改，其中 Resource 和 ResourceParams 元素用于配置数据库连接池：

```
<Context path="/17" docBase="E:\PublishBook\17">
     <Resource name="jdbc/ hibernateTest " scope="Shareable" type="javax.sql.DataSource"/>
     <ResourceParams name="jdbc/ hibernateTest ">
          <parameter>
               <name>factory</name>
               <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
          </parameter>

          <!--    数据库连接池信息    -->
          <parameter>
               <name>url</name>
               <value>jdbc:mysql://localhost/hibernateTest</value>
          </parameter>
<!---    数据库的 JDBC 驱动程序    --->
          <parameter>
               <name>driverClassName</name>
               <value>com.mysql.jdbc.Driver</value>
```

```
            </parameter>
<!---     下面是数据库的用户名和密码     --->
            <parameter>
                <name>username</name>
                <value>root</value>
            </parameter>
            <parameter>
                <name>password</name>
                <value>ict</value>
            </parameter>

            <!-- DBCP  连接池选项  -->
            <parameter>
                <name>maxWait</name>
                <value>3000</value>
            </parameter>
            <parameter>
                <name>maxIdle</name>
                <value>100</value>
            </parameter>
            <parameter>
                <name>maxActive</name>
                <value>10</value>
            </parameter>
        </ResourceParams>
</Context>
```

注意： 在使用上述配置文件时，不能包含中文的注释。

Tomcat 现在通过 JNDI 的方式：java:comp/env/jdbc/hibernateTest 来提供连接。如果遇到了 JDBC 驱动所报的 exception 出错信息，应该在没有 Hibernate 的环境下，先测试 JDBC 连接池本身是否配置正确。

注意： 应把数据库驱动程序 mysql-connector-java-3.0.16-ga-bin.jar 复制到 Web 应用的 WEB-INF\lib 目录下。

下面是一个用于测试这个连接池是否正确的一个简单的 JSP 页面，dataSourcetest.jsp 的代码如下：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import="java.sql.*,javax.sql.*,javax.naming.*"%>
<%

        Connection conn=null;
        try{
            InitialContext ctx = new InitialContext();
//查找并获取数据源
            DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/hibernateTest");
//获取数据库连接
            conn = ds.getConnection();
            if(conn!=null){
```

```
                out.println("数据源 jdbc/hibernateTest 配置正确！");
            }
        }catch(Exception ex){
            out.println("数据源 jdbc/hibernateTest 配置出现错误！"+ex);
        }
%>
```

把这个文件复制到 Web 应用的根目录下，然后在浏览器地址栏中输入如下地址：http://localhost:8080/17/dataSourcetest.jsp，这时如果页面显示如图 17.4 所示，则表明数据库连接池配置正确了。

注意： 关于配置文件的编写请参考 Servlet/JSP 标准文档。

图 17.4　验证数据源配置

17.1.4　编写持久化类

Hibernate 使用简单的 Java 对象（Plain Old Java Objects）这种编程模型来进行持久化。一个 POJO 很像 JavaBeans，它通过 getter 和 setter 方法访问其属性，对外则隐藏了内部实现的细节（假如需要，Hibernate 也可以直接访问其属性字段）。

下面是本例中使用的持久化类 Product 的源代码，由于它是一个很普通的 JavaBeans，这里不作更多介绍。

```
package cn.ac.ict;

import java.io.Serializable;
import java.util.Date ;

public class Product implements Serializable{
//下面是这个 JavaBeans 的属性
    private String pid ;
    private String pname;
    private String pcomp;
    private String pmadew;
    private Date pmadeyear;
    private float realprice;
    private float cutprice;
    private int amount;
    private String description;

    public Product(){

    }
//下面是各个属性的获取方法
    public String getPid(){
        return pid;
    }
    public String getPname(){
```

```
            return pname;
        }
        public String getPcomp(){
            return pcomp;
        }
        public String getPmadew(){
            return pmadew;
        }
        public Date getPmadeyear(){
            return pmadeyear;
        }
        public float getRealprice(){
            return realprice;
        }
        public float getCutprice(){
            return cutprice;
        }
        public int getAmount(){
            return amount;
        }
        public String getDescription(){
            return description;
        }

//下面是各个属性的设置方法
        public void setPid(String productid){
            pid = productid;
        }
        public void setPname(String productname){
            pname = productname;
        }
        public void setPcomp(String productcomp){
            pcomp = productcomp;
        }
        public void setPmadew(String productw){
            pmadew = productw;
        }
        public void setPmadeyear(Date madeyear){
            pmadeyear = madeyear;
        }
        public void setRealprice(float price){
            realprice = price;
        }
        public void setCutprice(float price){
            cutprice = price;
        }
        public void setAmount(int pamount){
            amount = pamount;
        }
```

```
        public void setDescription(String descr){
                description = descr;
        }

}
```

Hibernate 对属性使用的类型不加任何限制。所有的 Java JDK 类型和原始类型（比如 String、char 和 Date）都可以被映射，也包括 Java 集合框架（Java Collections Framework）中的类。可以把它们映射成为值、值集合，或者与其他实体类相关联。id 是一个特殊的属性，代表了这个类的数据库标识符（主键），对类似于 Cat 这样的实体类建议使用。Hibernate 也可以使用内部标识符，但这样会失去一些程序架构方面的灵活性。

持久化类不需要实现什么特别的接口，也不需要从一个特别的持久化根类继承下来。Hibernate 也不需要使用任何编译器处理，比如字节码增强操作，它独立地使用 Java 反射机制和运行时类增强（通过 CGLIB）。所以不依赖于 Hibernate，就可以把 POJO 的类映射成为数据库表。

17.1.5　编写 Hibernate 配置文件

Hibernate 中的配置文件可以是 properties 文件形式的，也可以是 XML 文件形式的，具体的编写方法在后面的小节中会详细介绍。本例中使用 XML 文件形式的配置文件，这个 XML 配置文件必须放在上下文类路径（WEB-INF/classes）下面，命名为 hibernate.cfg.xml（固定命名），完整代码如下：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="connection.datasource">java:comp/env/jdbc/hibernateTest</property>
        <property name="show_sql">false</property>
  <!--   使用 MySQL 数据库用语   -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <!--   映射文件   -->
        <mapping resource="Product.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

在这个配置文件中关闭了 SQL 命令的 Log，同时告诉 Hibernate 使用 MySQL 数据库用语（Dialet），以及如何得到 JDBC 连接（通过 Tomcat 声明绑定的 JNDI 地址）。Dialet 是

必须配置的，因为不同的数据库都和“SQL 标准”有一些差异。Hibernate 会根据这个 Dialet 处理这些差异，Hibernate 支持所有主流的商业和开放源代码数据库。

SessionFactory 是 Hibernate 中的一个概念，表示对应一个数据存储源。通过创建多个 XML 配置文件并在程序中创建多个 Configuration 和 SessionFactory 对象，就可以支持多个数据库了。

在 hibernate.cfg.xml 中的最后一个元素声明了 Product.hbm.xml，这是一个 Hibernate XML 映射文件，对应于持久化类 Product。这个文件包含了把 Product POJO 类映射到数据库表（或多个数据库表）的元数据。

17.1.6　编写映射文件

Hibernate XML 映射文件对应于持久化类，它的作用就是把持久化类的属性和数据表中的字段进行映射，下面是映射文件 Product.hbm.xml 的完整代码，这个文件必须放在上下文类路径（WEB-INF/classes）下面，文件的名字要与 Hibernate 配置文件中指定的映射文件名一致。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
<!----下面指定的类名必须是完全限定的类名，对应的数据表是 productlist -----   -->
    <class name="cn.ac.ict.Product" table="productlist">
<!------数据表的 ID，对应数据表中的 pid 项---   -->
        <id name="pid" type="string" unsaved-value="null" >
            <column name="pid" sql-type="char(15)" not-null="true"/>
        </id>
<!------下面是 JavaBeans 中属性与数据表中列的映射---   -->
        <property name="pname">
            <column name="pname" sql-type="char(30)"/>
        </property>

        <property name="pcomp">
            <column name="pcomp" sql-type="char(30)"/>
        </property>

        <property name="pmadew">
            <column name="madew" sql-type="char(20)"/>
        </property>

        <property name="pmadeyear" type="date" >
            <column name="madedate"/>
        </property>

        <property name="realprice" type="float" >
            <column name="rprice"/>
```

```
        </property>

        <property name="cutprice" type="float" >
            <column name="cprice"/>
        </property>

        <property name="amount" type="integer" >
            <column name="count" not-null="true"/>
        </property>

        <property name="description" type="string" >
            <column name="description" sql-type="char(100)"/>
        </property>
    </class>
</hibernate-mapping>
```

映射文件的根元素是<hibernate-mapping>，其可允许的子元素在 Hibernate 中都有规定，在 17.1.7 节中将对这些元素进行介绍。

17.1.7　编写 JSP 应用文件

上面的准备工作做完之后，就可以开始 Hibernate 的 Session 了。Session（与 JSP 和 Servlet 中的 Session 不一样）是一个持久化管理器，通过它从数据库中存取 Product 的对象的数据。

首先，要从 SessionFactory 中获取一个 Session（Hibernate 的工作单元）。

```
SessionFactory sessionFactory =
new Configuration().configure().buildSessionFactory();
```

通过对 configure()的调用来装载 hibernate.cfg.xml 配置文件，并初始化成一个 Configuration 实例。

在创建 SessionFactory 之前（它是不可变的），可以访问 Configuration 来设置其他属性（甚至修改映射的元数据）。应该在哪里创建 SessionFactory？在程序中又如何访问它呢？

SessionFactory 通常只是被初始化一次，例如通过一个 load-on-startup 的 Servlet 来初始化。这意味着开发者不应该在 Serlvet 中把它作为一个实例变量来持有，而应该放在其他地方。进一步地说，需要使用单例（Singleton）模式，开发者才能更容易地在程序中访问 SessionFactory。下面的方法就同时解决了两个问题：对 SessionFactory 的初始配置与便捷使用以及编写一个辅助的类文件 HibernateUtil.java：

```
package cn.ac.ict;

import org.hibernate.*;
import org.hibernate.cfg.*;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class HibernateUtil {
```

```
    private static Log log = LogFactory.getLog(HibernateUtil.class);

    private static final SessionFactory sessionFactory;

    static {
        try {
            // 创建 SessionFactory
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            //输出日志信息
            log.error("Initial SessionFactory creation failed.", ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession()   {
        Session s = (Session) session.get();
        // 打开一个新的会话
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }
//关闭一个 Hibernate 会话
    public static void closeSession() {
        Session s = (Session) session.get();
        if (s != null)
            s.close();
        session.set(null);
    }
}
```

这个类不但在它的静态初始器中使用了 SessionFactory，还使用了一个 ThreadLocal 变量来保存 Session 作为当前工作线程。

SessionFactory 是安全线程，可以由很多线程并发访问并获取 Session。单个 Session 不是安全线程对象，它只代表与数据库之间的一次操作。Session 通过 SessionFactory 获得并在所有的工作完成后关闭。

在一个 Session 中，每个数据库操作都是在一个事务（Transaction）中进行的，这样就可以隔离开不同的操作（甚至包括只读操作）。使用 Hibernate 的 Transaction API 从底层的事务策略中（本例中是 JDBC 事务）“脱身”出来。这样，开发者不需要更改任何源代码，就可以把程序部署到一个由容器管理事务的环境中去（使用 JTA）。

这样可以随心所欲地多次调用 HibernateUtil.currentSession()，而且每次都会得到当前线程的同一个 Session。不论是在 Servlet 代码中或者在 Servlet Filter 中还是在 HTTP 结果返回

之前，开发者都必须确保这个 Session 在数据库访问工作完成后关闭。这样做的一个好处就是可以容易地使用延迟装载（Lazy Initialization）。

下面是获得一个 Session 然后进行数据更新的一个简单 JSP 页面（add.jsp）：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import="org.hibernate.*,org.hibernate.cfg.*,cn.ac.ict.*,java.text.*"%>

<body>
<p class="style1">
<%
    if(request.getParameter("pid")!=null){
//获取多个请求参数
        String pid = request.getParameter("pid");
        String pname = request.getParameter("pname");
        String pcomp = request.getParameter("pcomp");
        String pmadew = request.getParameter("pmadew");
        java.util.Date pmadeyear = DateFormat.getDateInstance().parse(request.getParameter
("pmadeyear"));
        out.println(pmadeyear);
        float realprice =Float.parseFloat(request.getParameter("realprice"));
        float cutprice =Float.parseFloat(request.getParameter("cutprice"));
        int amount = Integer.parseInt(request.getParameter("amount"));
        String description = request.getParameter("description");

  try{
        Transaction tx=null;
//根据客户的请求信息构建一个持久化对象
        Product product = new Product();
        product.setPid(pid);
        product.setPname(pname);
        product.setPcomp(pcomp);
        product.setPmadew(pmadew);
        product.setPmadeyear(pmadeyear);
        product.setRealprice(realprice);
        product.setCutprice(cutprice);
        product.setAmount(amount);
        product.setDescription(description);
        //开始一个 Hibernate 会话
        Session Hsession = HibernateUtil.currentSession();
            tx= Hsession.beginTransaction();
//将持久化对象保存到数据库
            Hsession.save(product);
            tx.commit();
        out.print(product);

 }catch(Exception e){out.print("insert error!"+e); }

  }
%>
</p>
<p>Add   New Product:</p>
```

```
<!--------------下面是用于接收输入的表单------------------>
<form action="add.jsp" method="post" name="form1" target="_self">
  <p>Product ID:
    <input name="pid" type="text" id="pid">
  </p>
   <p>ProductName:
     <input name="pname" type="text" id="pname">
   </p>
   <p>Company:
     <input name="pcomp" type="text" id="pcomp">
  </p>
  <p>Made Where :
     <input name="pmadew" type="text" id="pmadew">
  </p>
  <p>Made Date ：
    <input name="pmadeyear" type="text" id="pmadeyear">
  </p>
  <p>Real Price：
    <input name="realprice" type="text" id="realprice">
</p>
  <p>Cut Price：
    <input name="cutprice" type="text" id="cutprice">
  </p>
  <p>Amount ：
    <input name="amount" type="text" id="amount">
  </p>
  <p>Description：
    <input name="description" type="text" id="description">
  </p>
  <p>
     <input type="submit" name="Submit" value="提交">
</p>
</form>
```

注意： 这个 JSP 文件对于读者不正确的输入不作处理，读者在输入数据时注意要依据数据类型来输入，如对于生产日期就应按照 YYYY-MM-DD 的格式输入。

17.1.8 编译并发布 Web 应用

读者可以按照如下步骤发布这个 Web 应用：

（1）编译 HibernateUtil 类，需要把 Hibernate 的核心包和 Hibernate 运行必需的包存放在类路径中。

（2）检查文件是否按照正确的位置存放：

- Hibernate 配置文件和映射文件存放在 Web 应用的 WEB-INF\classes 目录下。
- Hibernate 的核心包和 Hibernate 运行必需的包以及数据库驱动程序存放在 WEB-INF\lib 目录下。

（3）启动 Tomcat，在浏览器地址栏中输入如下地址：http://localhost:8080/17/add.jsp，

在显示的页面中输入数据，如图 17.5 所示。

（4）提交后，在 MySQL 数据库中查询结果，可以看到刚刚输入的信息都被保存到数据库中了，如图 17.6 所示。



图 17.5　产品信息

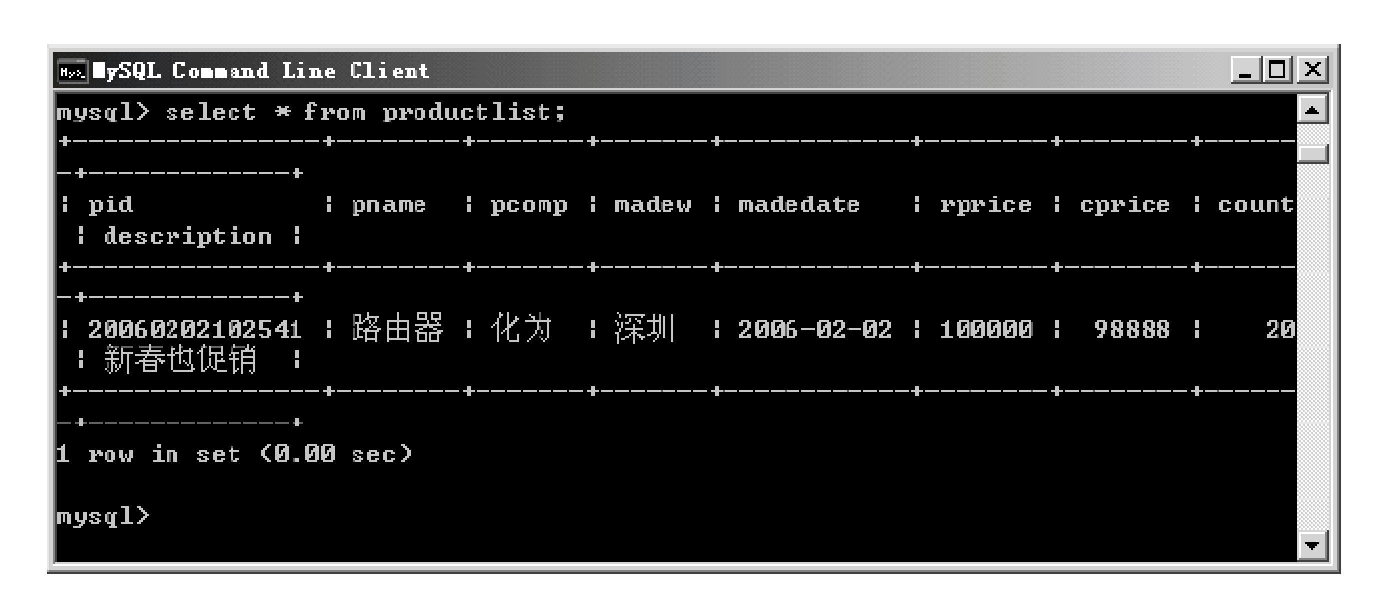


图 17.6　Hibernate 数据持久化

17.2　Hibernate 技术介绍

Hibernate 技术涉及很多方面，下面初步介绍几个方面，读者如果需要详细了解 Hibernate 技术，请参考 Hibernate 文档。

17.2.1　映射定义

Hibernate XML 映射文件把 Hibernate 持久类和数据库中的数据字段进行映射，本节将介绍编写 Hibernate XML 映射文件时使用的一些元素。

1．class 元素

使用 class 元素定义一个持久化类，下面是 class 元素的语法定义：

```
<class
        name="ClassName"
        table="tableName"
        discriminator-value="discriminator_value"
        mutable="true|false"
        schema="owner"
        catalog="catalog"
        proxy="ProxyInterface"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        select-before-update="true|false"
        polymorphism="implicit|explicit"
```

```
        where="arbitrary sql where condition"
        persister="PersisterClass"
        batch-size="N"
        optimistic-lock="none|version|dirty|all"
        lazy="true|false"
        entity-name="EntityName"
        check="arbitrary sql check condition"
        rowid="rowid"
        subselect="SQL expression"
        abstract="true|false"
        entity-name="EntityName"
        node="element-name"
/>
```

所有属性都是可选的，下面简单介绍几个属性的作用和使用方法：

- ❑ name（可选）：持久化类（或者接口）的完整类名（包括包名）。如果这个属性不存在，Hibernate 将假定这是一个非 POJO 的实体映射。
- ❑ table（可选，默认是类的非全限定名）：对应的数据库表名。

如例中可以看到持久化类的名字是 cn.ac.ict.Cat，在数据库中对应的表名是 cats。

注意： 类（或者接口）的 Java 完整类名是指包含类或接口所在包的名字，如例中 Product 类在 cn.ac.ict 包中，则其 Java 全限定名是 cn.ac.ict. Product，非全限定名是 Product。

2．id 元素

被映射的类必须定义对应数据库表的主键字段。大多数类有一个 JavaBeans 风格的属性，为每一个实例包含惟一的标识。<id>元素定义了该属性到数据库表的主键字段的映射。id 元素的语法定义如下：

```
<id
        name="propertyName"
        type="typename"
        column="column_name"
        unsaved-value="null|any|none|undefined|id_value"
        access="field|property|ClassName"
        node="element-name|@attribute-name|element/@attribute|.">
        <generator class="generatorClass"/>
</id>
```

其中：

- ❑ name（可选）：标识属性的名字。
- ❑ type（可选）：标识 Hibernate 类型的名字。
- ❑ column（可选，默认为属性名）：主键字段的名字。

注意： 如果 name 属性不存在，会认为这个类没有标识属性。

id 元素有一个可选的<generator>子元素，它是一个 Java 类的名字，用来为该持久化类的实例生成惟一的标识。如果这个生成器实例需要某些配置值或者初始化参数，可以用<param>元素来传递。例如：

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

注意： 标识的生成策略有很多种，具体使用的策略可以根据实际需要而定，读者如果需要了解可以查看 Hibernate 的参考文档。

3．property 元素

<property>元素为类定义一个持久化的 JavaBeans 风格的属性。<property>元素的语法定义如下：

```
<property
        name="propertyName"
        column="column_name"
        type="typename"
        update="true|false"
        insert="true|false"
        formula="arbitrary SQL expression"
        access="field|property|ClassName"
        lazy="true|false"
        unique="true|false"
        not-null="true|false"
        optimistic-lock="true|false"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

- name：属性的名字，以小写字母开头，与持久化类中的属性名字一致。
- column（可选－默认为属性名字）：对应的数据库字段名。也可以通过嵌套的<column>元素指定。
- type（可选）：一个 Hibernate 类型的名字。

Hibernate 类型可以是如下几种：

- Hibernate 基础类型之一（比如：integer、string、character、date、timestamp、float、binary、serializable、object 和 blob）。
- 一个 Java 类的名字，这个类属于一种默认基础类型（比如：int、float、char、java.lang.String、java.util.Date、java.lang.Integer 和 java.sql.Clob）。
- 一个可以序列化的 Java 类的名字。
- 一个自定义类型的类的名字（比如：cn.ac.ict.type.MyCustomType）。

如果不指定类型，Hibernarte 会使用映射来得到这个名字的属性，以此来猜测正确的 Hibernate 类型。Hibernate 会按照规则 2，3，4 的顺序对属性读取器（getter 方法）的返回类进行解释。

注意： 在某些情况下仍然需要 type 属性。比如，为了区别 Hibernate.DATE 和 Hibernate.TIMESTAMP，或者为了指定一个自定义类型。

17.2.2　Hibernate 的类型

1．基本值类型

Hibernate 内建的基本映射类型可以大致分为如下几类：

❑ Integer、long、short、float、double、character、byte、boolean、yes_no、true_false

这些类型都对应 Java 的原始类型或者其封装类，来符合（特定厂商的）SQL 字段类型。boolean、yes_no 和 true_false 都是 Java 中 boolean 或者 java.lang.Boolean 的另外说法。

❑ string

从 java.lang.String 到 VARCHAR（或者 Oracle 的 VARCHAR2）的映射。

❑ date、time、timestamp

从 java.util.Date 和其子类到 SQL 类型 DATE、TIME 和 TIMESTAMP（或等价类型）的映射。

❑ calendar、calendar_date

从 java.util.Calendar 到 SQL 类型 TIMESTAMP 和 DATE（或等价类型）的映射。

❑ big_decimal、big_integer

从 java.math.BigDecimal 和 java.math.BigInteger 到 NUMERIC（或者 Oracle 的 NUMBER 类型）的映射。

❑ locale、timezone、currency

从 java.util.Locale、java.util.TimeZone 和 java.util.Currency 到 VARCHAR（或者 Oracle 的 VARCHAR2 类型）的映射，locale 和 currency 的实例被映射为它们的 ISO 代码。timezone 的实例被影射为它的 ID。

❑ class

从 java.lang.Class 到 VARCHAR（或者 Oracle 的 VARCHAR2 类型）的映射。class 被映射为它的全限定名。

❑ binary

把字节数组（Byte Arrays）映射为对应的 SQL 二进制类型。

❑ text

把长 Java 字符串映射为 SQL 的 CLOB 或者 TEXT 类型。

❑ serializable

把可序列化的 Java 类型映射到对应的 SQL 二进制类型。也可以为一个并非默认为基本类型的可序列化 Java 类或者接口指定 Hibernate 类型 serializable。

❑ clob、blob

JDBC 类 java.sql.Clob 和 java.sql.Blob 的映射。某些程序可能不适合使用这个类型，因为 blob 和 clob 对象可能在一个事务之外是无法重用的。

在 org.hibernate.Hibernate 中，定义了基础类型对应的 Type 常量。比如，Hibernate.STRING 代表 string 类型。

2．自定义值类型

开发者创建属于他们自己的值类型也是很容易的。比如说，开发者可能希望持久化一个 java.lang.BigInteger 类型的属性，使其持久化成为 VARCHAR 字段。Hibernate 没有内置这样一种类型。自定义类型能够映射一个属性（或集合元素）到不止一个数据库表字段。比如说，可能有这样的 Java 属性：getName()/setName()是 java.lang.String 类型的，对应的持久化到 3 个字段：FIRST_NAME、INITIAL 和 SURNAME。

要实现一个自定义类型，可以实现 org.hibernate.UserType 或 org.hibernate.CompositeUserType 中的任一个，并且使用类型的 Java 全限定类名来定义属性。

17.2.3　Hibernate 事务

Hibernate 是对 JDBC 的轻量级对象封装，Hibernate 本身是不具备 Transaction 处理功能的，Hibernate 的 Transaction 实际上是底层的 JDBC Transaction 的封装，或者是 JTA Transaction 的封装，Hibernate 可以配置为 JDBCTransaction 或者是 JTATransaction，这取决于在 Hibernate 配置文件中的配置：

```
#hibernate.transaction.factory_class net.sf.hibernate.transaction.JTATransactionFactory
#hibernate.transaction.factory_class net.sf.hibernate.transaction.JDBCTransactionFactory
```

如果什么都不配置，默认情况下使用 JDBCTransaction，如果配置为：

```
hibernate.transaction.factory_class net.sf.hibernate.transaction.JTATransactionFactory
```

将使用 JTATransaction。

17.3　Hibernate 配置

Hibernate 使用数据库和配置信息来为应用程序提供持久化服务，它实现与数据库连接的过程如图 17.7 所示。

由于 Hibernate 是为了能在各种不同环境下工作而设计的，因此存在着大量的配置参数，Hibernate 得到配置信息存在很多种方式。

- 编程的配置方式。
- 传一个 java.util.Properties 实例给 Configuration.setProperties()。
- 将 hibernate.properties 放置在类路径（classpath）的根目录下（root directory）。
- 通过 java -Dproperty=value 来设置系统（System）属性。
- 在 hibernate.cfg.xml 中加入元素<property>。

其中使用 hibernate.properties 文件是最简单的方式，下面结合这几种方式讲述 Hibernate 是如何实现数据库连接的。

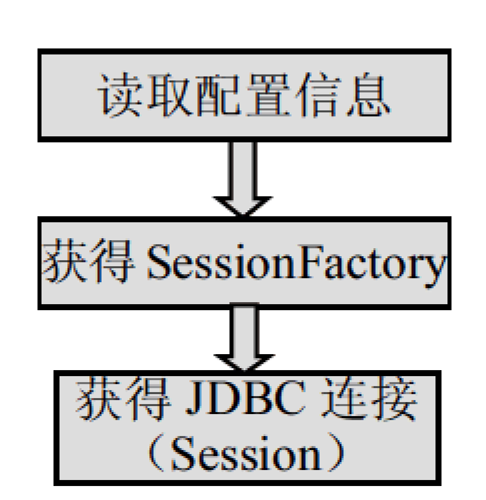


图 17.7　Hibernate 获得数据库连接

一个 org.hibernate.cfg.Configuration 实例代表了一个应用程序中 Java 类型到 SQL 数据库映射的完整集合。Configuration 被用来构建一个（不可变的）SessionFactory。

17.3.1　可编程的配置方式

可编程的配置方式并不是最好的方式，但通过了解可编程的配置方式，可以对 Hibernate 的配置过程的实现有更好的理解，而且通过这种方式来了解 Hibernate 的一些配置属性的作用和使用方法。

1．获得配置属性

编程的配置方式是直接实例化 Configuration 来获取一个实例，并为它指定 XML 映射定义文件。如果映射定义文件在类路径（classpath）中，就使用 addResource()方法把 XML 映射定义文件的信息加载从而进行 Hibernate 的配置。

```
Configuration cfg = new Configuration().addResource("Item.hbm.xml")
                    .addResource("Bid.hbm.xml");
```

一种更好的方式是指定被映射的类，让 Hibernate 寻找映射定义文件：

```
Configuration cfg = new Configuration().addClass(org.hibernate.auction.Item.class).addClass(org.
hibernate.auction.Bid.class);
```

Hibernate 将会在类路径中寻找名字为/org/hibernate/auction/Item.hbm.xml 和/org/hibernate/auction/Bid.hbm.xml 映射定义文件。这种方式消除了任何对文件名的硬编码。

Configuration 也允许开发者编码指定配置属性：

```
Configuration cfg = new Configuration()
     .addClass(org.hibernate.auction.Item.class)
     .addClass(org.hibernate.auction.Bid.class)
     .setProperty("hibernate.dialect",
"org.hibernate.dialect.MySQLInnoDBDialect")
     .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
     .setProperty("hibernate.order_updates", "true");
```

Configuration 实例是一个启动期间的对象，一旦它完成创建 SessionFactory 的任务，它就被丢弃了。

2．获得 SessionFactory

当所有映射定义被 Configuration 解析后，应用程序必须获得一个用于构造 Session 实例的工厂。这个工厂将被应用程序的所有线程共享：

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate 允许应用程序创建多个 SessionFactory 实例。这对使用多个数据库的应用来说很有用。

3．JDBC 连接

通常希望 SessionFactory 来创建和缓存（数据库连接池）JDBC 连接。如果采用这种方

式，只需要如下所示那样，打开一个 Session：

```
// 打开一个新的 Session
Session session = sessions.openSession();
```

一旦需要进行数据访问时，就会从连接池中获取一个 JDBC 连接。为了使这种方式工作起来，需要向 Hibernate 传递一些 JDBC 连接的属性。所有 Hibernate 属性的名字和语义都在 org.hibernate.cfg.Environment 中定义。下面将描述 JDBC 连接配置中最重要的几项设置。

如果设置如表 17.2 所示的属性，Hibernate 将使用 java.sql.DriverManager 来获得（和缓存）JDBC 连接。

表 17.2　Hibernate JDBC属性

属　性　名	用　　途
hibernate.connection.driver_class	JDBC驱动类
hibernate.connection.url	JDBC URL
hibernate.connection.username	数据库用户
hibernate.connection.password	数据库用户密码
hibernate.connection.pool_size	连接池容量上限数目

但 Hibernate 自带的连接池算法相当不成熟。它只是为了让初学者快些上手，不适合用于产品系统或性能测试中。出于最佳性能和稳定性考虑应该使用第三方的连接池。

连接池的特定设置替换 hibernate.connection.pool_size，这将关闭 Hibernate 自带的连接池。例如，可以考虑使用 Tomcat 的连接池。

为了能在应用程序服务器（Application Server）中使用 Hibernate，应当先将 DataSource 在 JNDI 中注册，然后让 Hibernate 从 DataSource 处获得连接，这至少需要设置下列属性（如表 17.3 所示）中的一个。

表 17.3　Hibernate数据源属性

属　性　名	用　　途
hibernate.connection.datasource	数据源JNDI名字
hibernate.jndi.url	JNDI提供者的URL（可选）
hibernate.jndi.class	JNDI InitialContextFactory类（可选）
hibernate.connection.username	数据库用户（可选）
hibernate.connection.password	数据库用户密码（可选）

下面是一个使用应用程序服务器 JNDI 数据源的 hibernate.properties 样例文件：

```
#数据源 JNDI 名字
hibernate.connection.datasource = java:/comp/env/jdbc/test
#工厂类
hibernate.transaction.factory_class = \
org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

从 JNDI 数据源获得的 JDBC 连接将自动参与应用程序服务器中容器管理的事务中去。

任何连接配置属性的属性名要以“hibernate.connection”前缀开头。例如，可以使用 hibernate.connection.charSet 来指定 charSet。

通过实现 org.hibernate.connection.ConnectionProvider 接口，可以定义属于自己的获得 JDBC 连接的插件策略。通过设置 hibernate.connection.provider_class，可以选择一个自定义的实现。

注意： 除了以上属性外还有大量属性能用来控制 Hibernate 在运行期的行为，它们都是可选的，并拥有适当的默认值，这里不详细介绍。

17.3.2　XML 配置文件方式

另一个配置方法是在 hibernate.cfg.xml 文件中指定一套完整的配置。这个文件可以当成 hibernate.properties 的替代，它使用标准的 XML 语法，是比较受欢迎的一种方式。

下面是一个配置文件的例子：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!--  数据库连接设置 -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/hibernate_test</property>
        <property name="connection.username">root</property>
        <property name="connection.password">ict</property>

        <!-- JDBC 连接池（使用内置的连接池）-->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <!--- 显示所有的 SQL 语句到标准输出 -->
        <property name="show_sql">true</property>

        <!--   启动时删除并新建数据库 schema -->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource="Student.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

注意： 在 XML 配置文件中使用的属性与前面介绍的属性名字是基本一样的，惟一的差别就在于 XML 配置文件中的属性名省略了 hibernate 前缀。

使用 XML 配置使得启动 Hibernate 变得非常简单，而且还具有 XML 语法的优势。如下所示，一行代码就可以获得一个 SessionFactory 实例：

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

另外可以使用如下代码来添加一个不同的 XML 配置文件：

```
SessionFactory sf = new Configuration().configure("catdb.cfg.xml").buildSessionFactory();
```

17.4　小　　结

Hibernate 是一个 JDO 工具。它的工作原理是通过文件将值对象和数据库表之间建立起一个映射关系，这样，开发者只需要通过操作这些值对象和 Hibernate 提供的一些基本类，就可以达到使用数据库的目的。

在本章中简单介绍了 Hibernate 的使用和一些基本的知识，读者要学好 Hibernate，就应该努力搞懂关系映射的原理，多实践体会。

第 18 章　JSP Web 应用的设计概述

软件应用的设计是一个在软件开发中很重要的环节，设计不好，往往使前面的工作变得没有意义。在软件设计的过程中不但要考虑设计模式的问题，而且还需要考虑调试、测试、安全等方面的问题，在本章中将简单介绍 JSP Web 应用的设计和实践方面的一些问题，读者可以根据这些知识更好地学习软件设计。

18.1　可维护性与可扩展性设计

当谈到设计时，读者并不一定就会想到是软件的设计，因为在其他很多行业都有设计，事先考虑好相关的因素会使得设计更加简单化，例如要设计一个厨房，就要事先考虑好水管、电线、窗户和门的位置，如果不能事先考虑好这些因素，就会在实际使用时碰到很多问题，这个道理同样适合于软件的设计，在设计时就要对要达成的目标和将来可能做的改动有一定的了解，并在设计时为它们留出足够的空间。

软件工业与其他工业的某些形式是很相似的，尤其是建筑行业。虽然这种比较应该在不同的层次上进行，它们之间还是有一些不同的。通常建筑设计的目的是很确定的，它必须能够经受环境的已知程度的变化，但软件的设计就不像建筑设计那样准确，它显得更加动态化，在软件的生命周期中，可能涉及一些更改，有时可能只是消除一些小缺陷，而更多地可能是软件功能的变更和增加，这些变更是决定软件设计特点的重要原因，一般在软件设计的过程中需要考虑软件的可维护性、可扩展性、可重用性以及健壮性等，下面简单介绍可维护性和可扩展性。

18.1.1　可维护性

简单而言，可维护性是指一个软件在业务逻辑（包括需求）、实现技术改变时是否可以被轻松地修改。跨越时间的复用和易维护性很相近，但前者更多地侧重如何完全不动地使用，而后者则是侧重让改动的影响更小。

对业务单一的公司而言，可维护性是软件仅次于稳定性的第二属性；对多业务公司而言，或许要排在可复用性之后。

当软件易维护性高时，意味着可以更快地适应业务逻辑变化，这对于新兴的小公司而言经常是生存的依靠。因为小公司比之大公司，更可能生产频繁升级的甚至是用户定制的软件，这对于那些说不清楚自己想要什么的用户而言具有很大吸引力。

当软件易维护性高时，还可以很容易地采用新技术，这将更有利于将产品推到市场前

沿，保持竞争力和增进技术积累，这对于根基已稳的大公司而言是一个长远战略。有很多大型产品（如数据库产品），其业务逻辑很少变动，改进技术是主要的工作。

18.1.2　可扩展性

软件系统的另外一个很重要的方面是它的可扩展性，也就是对其进行扩展和增强的能力。一个设计良好的软件应该有一个预定义好的框架，使得已有的特性可以很容易地被修改，而新的特性又可以很容易地被加入，并且不会对系统的其他部分产生副作用，尤其对于那些以任务为中心的商业系统，可扩展性是软件的一个很重要的因素，它可以使软件保持和商业处理同步，从而降低了费用。

18.2　JSP Web 应用的设计

在对软件设计方面有了一些了解后，下面开始介绍基于 Java 的两种常用设计：以页面为中心的设计和 MVC 设计。

18.2.1　以页面为中心的设计（Model 1）

第一个构建软件的通用架构是以页面为中心的设计也是 Web 应用程序集合在一起的最简单形式，在这种设计中，只需要关注 JSP 页面的编写，其基本的描述如图 18.1 所示。

在使用 Java 技术建立 Web 应用的实例中，由于 JSP 技术的发展，很快这种便于掌握和可实现快速开发的技术就成了创建 Web 应用的主要技术。JSP 页面中可以非常容易地结合业务逻辑（jsp:useBean）、服务端处理过程（jsp:scriptlet）和 HTML（<html>），在 JSP 页面中同时实现显示，业务逻辑和流程控制，从而可以快速地完成应用开发。现在很多的 Web 应用就是由一组 JSP 页面构成的。这种以 JSP 为中心的开发模型可以称之为 Model 1。

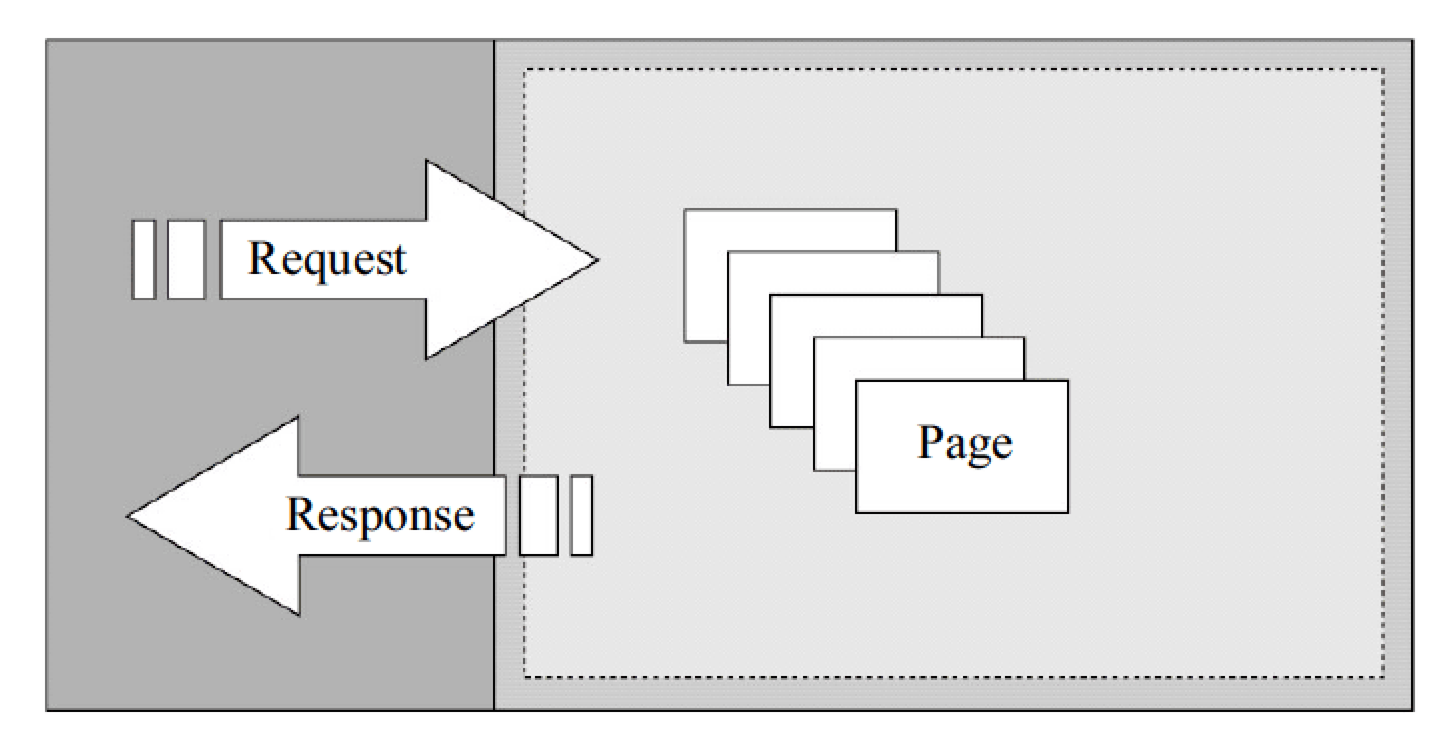


图 18.1　Model 1 架构

在这种设计模式时，有两种方法可以实现数据的共享，一种是把所有访问数据库的 SQL 语句写在 JSP 页面中来连接数据库并得到返回的结果；另一种是把访问数据库的代码单独放到一个 Java 类或者自定义的标签中，例如 JSTL 中的一些 SQL 标签。这种方式是 JSP Web 设计模式中最不具有可维护性的了，因为，一旦数据库的访问策略变了，就需要把所有需要访问数据库的页面进行修改。

另一种在以页面为中心的应用中访问数据的方式是使用 JavaBeans，使用 JavaBeans 表示系统中需要持久化的实体，如图 18.2 所示。例如，开发者可以使用一个 Product 对象封装系统中要处理的产品信息，这样访问数据库的代码就从 JSP 页面中分离出来，被放入了可重用的 JavaBeans 类中，这样就可以在获取应用的高可维护性的同时也获取了代码的可重用性，数据库的访问策略变化时只需要更改几个 JavaBeans 类。

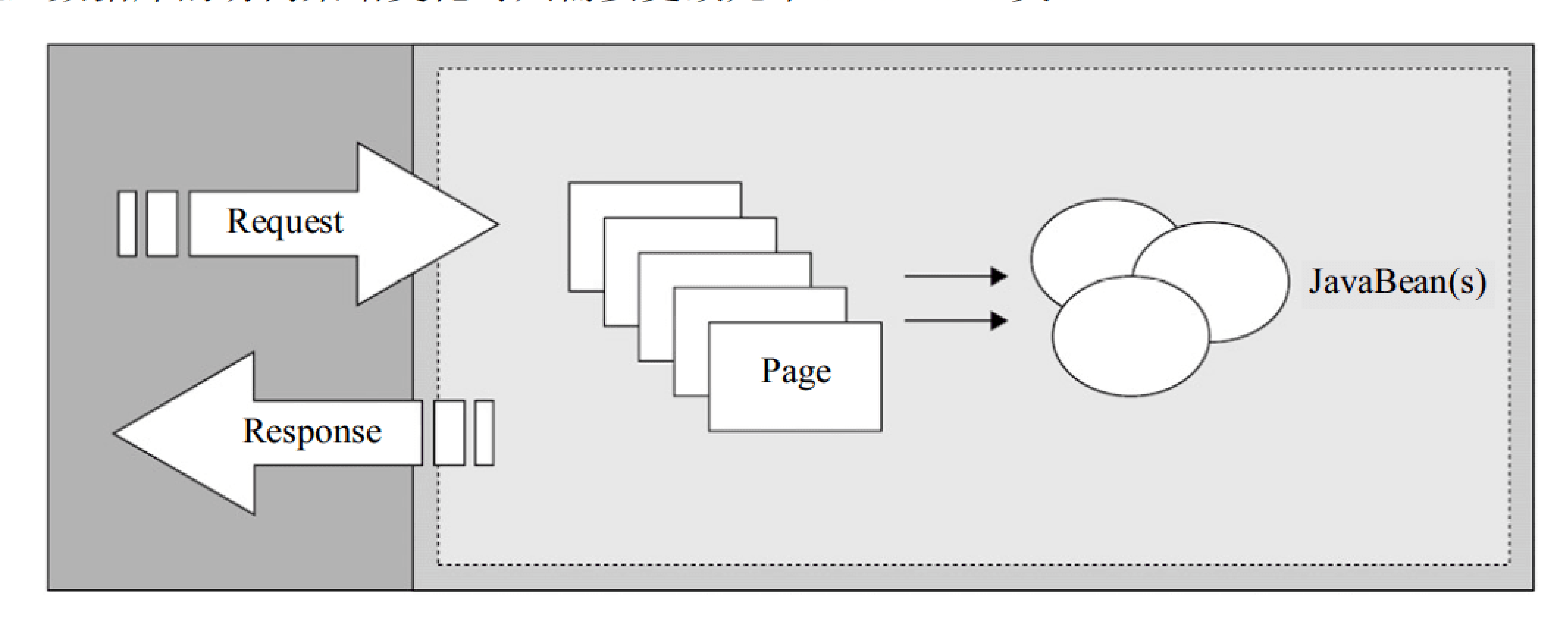


图 18.2　使用 JavaBeans 访问数据库的 Model 1

这种开发模式在进行快速和小规模的应用开发时有非常大的优势，但从工程化的角度考虑，它即使结合 JavaBeans 技术后还是有一些不足之处：

- 应用的实现一般是基于过程的，一组 JSP 页面实现一个业务流程，如果要进行改动，必须在多个地方进行修改。这样非常不利于应用扩展和更新。
- 由于应用不是建立在模块上的，业务逻辑和表示逻辑混合在 JSP 页面中没有进行抽象和分离，所以非常不利于应用系统业务的重用和改动。

考虑到这些问题，在开发大型的 Web 应用时必须采用不同的设计模式——Model 2。

18.2.2　MVC 设计模式（Model 2）

模型－视图－控制器（MVC）是 Xerox PARC 在 20 世纪 80 年代为编程语言 Smalltalk－80 发明的一种软件设计模式，最近几年被推荐为 Sun 公司 J2EE 平台的设计模式，并且受到越来越多的使用 ColdFusion 和 PHP 的开发者的欢迎。MVC 设计模式包括 3 类对象：

- 模型（Model）对象：是应用程序的主体部分。
- 视图（View）对象：是应用程序中负责生成用户界面的部分。
- 控制器（Controller）对象：是根据用户的输入，控制用户界面数据显示及更新 Model 对象状态的部分。下面介绍它们之间的关系和各自的主要功能。

视图（View）代表用户交互界面，对于 Web 应用来说，可以概括为 HTML 界面、XHTML、XML 和 Applet。MVC 设计模式对于视图的处理仅限于视图上数据的采集和处理，以及用户的请求，而不包括在视图上的业务流程的处理。业务流程的处理交给模型（Model）处理。比如一个订单的视图只接受来自模型的数据并显示给用户，以及将用户界面的输入数据和请求传递给控制和模型，MVC 设计模式基本结构如图 18.3 所示。

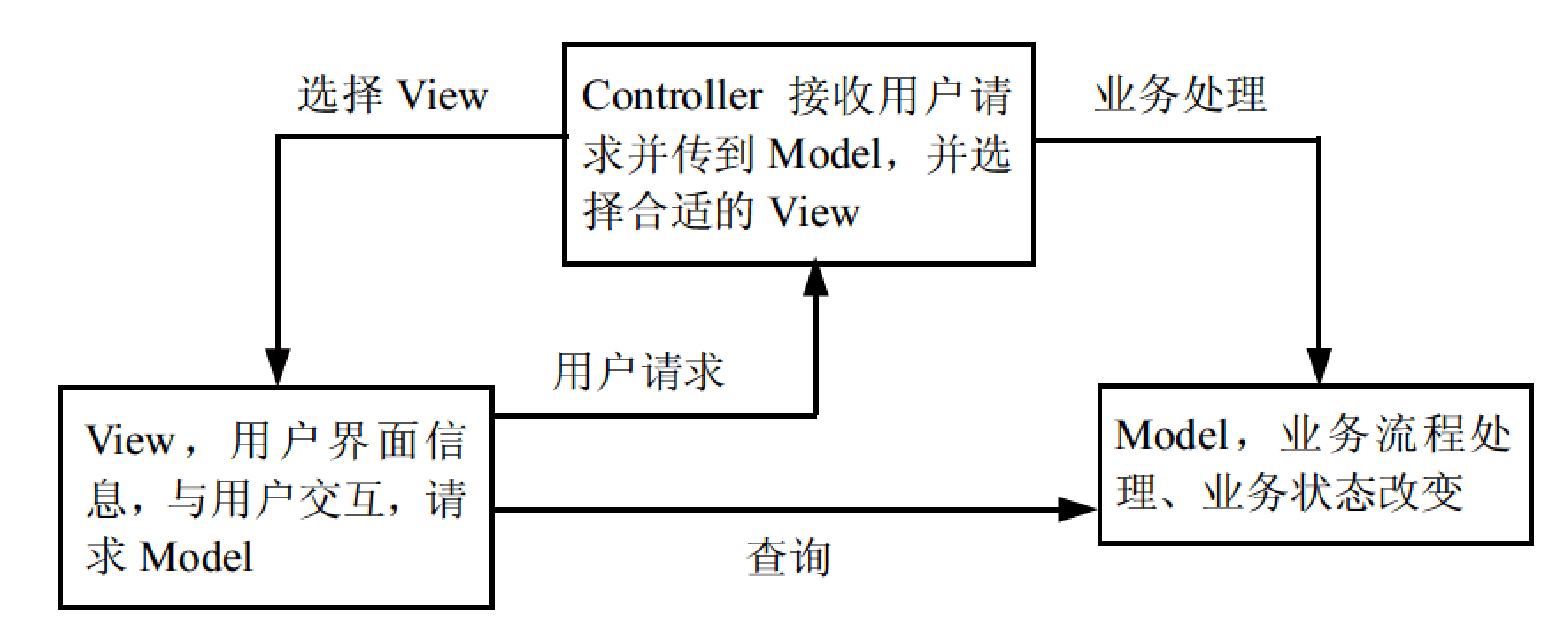


图 18.3　MVC 设计模式基本结构

模型（Mode）就是业务流程/状态的处理以及业务规则的指定。业务流程的处理过程对其他层来说是黑箱操作，模型接收视图请求的数据，并返回最终的处理结果。业务模型的设计可以说是 MVC 最主要的核心。

业务模型还有一个很重要的模型就是数据模型。数据模型主要指实体对象的数据保存（持续化）。比如将一张订单保存到数据库，从数据库获取订单。可以将这个模型单独列出，所有有关数据库的操作只限制在该模型中。

控制（Controller）从用户接收请求，将模型与视图匹配在一起，共同完成用户的请求。划分控制层的作用也很明显，它就是一个分发器，选择什么样的模型，选择什么样的视图，可以完成什么样的用户请求。控制层并不作任何的数据处理。例如，用户单击一个连接，控制层接受请求后，并不处理业务信息，它只把用户的信息传递给模型，告诉模型做什么，选择符合要求的视图返回给用户。因此，一个模型可能对应多个视图，一个视图可能对应多个模型。

MVC 模式不仅实现了功能模块和显示模块的分离，同时它还提高了应用系统的可维护性、可扩展性、可移植性和组件的可复用性。

18.3　Web 应用的架构框架

随着网络应用的快速增加，MVC 模式对于 Web 应用的开发无疑是一种非常先进的设计思想，无论选择哪种语言，无论应用多复杂，它都能在理解分析应用模型时提供最基本的分析方法，构造产品提供清晰的设计框架，为软件工程提供规范的依据。

现在已经有了很多的 MVC 模式，例如 Struts、Tapestry、WebWork2 等，不同的实现在具体的策略上当然是有所不同，本节将简单介绍几个 MVC 模式的实现，在后面的章节中还会继续介绍如何使用它们进行开发。

18.3.1　Struts——最流行的 MVC 框架

Struts 是 Apache Jakarta 项目的组成部分。该项目的目标是为建立 Java Web 应用程序而提供的一个开源框架，目前一般使用的版本为 1.1，最新版本是 1.2。通过使用 Struts 框架可以改进和提高 Java Server Pages（JSP）、Servlet、标签库以及面向对象的技术在 Web 应用

程序中的应用。应用 Struts 框架可以减少应用 MVC（Model-View-Controller）设计模式的开发时间，从而提高开发效率。

Struts 是 MVC 的一种实现，它很好地结合了 Jsp、Java Servlet、JavaBeans 和 Taglib 等技术，是 MVC 模式的经典实现。

18.3.2　WebWork2——基于 XWork 的 MVC 框架

WebWork 是由 OpenSymphony 组织开发的，是致力于组件化和代码重用的拉出式 MVC 模式 J2EE Web 框架。WebWork 目前最新版本是 2.1，现在的 WebWork2.x 前身是 Rickard Oberg 开发的 WebWork。WebWork2 建立在 XWork 之上处理 HTTP 的响应和请求。

18.3.3　Spring——以控制倒置原则为基础的 MVC 框架

Spring 是一个以控制倒置（Inversion of Control）原则为基础的轻量级框架。控制倒置是一个用于“基于组件的体系结构”的设计模式，它将“判断依赖关系”的职责移交给容器，而不是由组件本身来判断彼此之间的依赖关系。当在 Spring 内实现组件时，容器“轻量级”的方面就展现出来了：针对 Spring 开发的组件不需要任何外部库；而且，容器是轻量级的，它避免了像 EJB 容器那样的重量级方案的主要缺点，例如启动时间长、测试复杂、部署和配置困难等。

18.3.4　Java Server Faces——Sun 力推的 MVC 框架

Java Server Faces（JSF）是一种 MVC 应用程序框架，用于创建基于 Web 的用户界面。如果读者熟悉 Struts（一种流行的开放源代码的基于 JSP 的 Web 应用程序框架）和 Swing（针对桌面应用程序的标准 Java 用户界面），就可以将 Java Server Faces 想象成这两种框架的组合。与 Struts 一样，JSF 通过一个控制器 Servlet 来提供 Web 应用程序生命周期管理；也与 Swing 一样，JSF 提供了一个带有事件处理和组件呈现（rendering）的丰富组件模型。

18.4　Web 应用的测试

在一个软件开发项目中，软件的测试是一个必不可少的工作，为了保证工程的质量需要，对软件进行的测试有：功能测试、性能测试、安全性测试、稳定性测试、浏览器兼容性测试等多项测试。其中功能测试又是最基本的一项测试，它是其他测试的基础。

18.4.1　JUnit——优秀的单元测试工具

对于 Java 程序而言，JUnit 是一个非常优秀的单元测试工具，可以进行有效的功能测试，

JUnit 是当前 Java 语言单元测试的一站式解决方案，它把测试驱动的开发思想介绍给 Java 开发人员并教给他们如何有效地编写单元测试。众多的优点使得它成为一种优秀的测试工具，在本章就介绍如何使用 JUnit 进行 Java 的单元测试。

18.4.2　Cactus——基于 JUnit 框架的服务器端测试工具

Cactus 是一个基于 JUnit 框架的简单测试框架，用来单元测试服务端 Java 代码。Cactus 框架的主要目标是能够单元测试服务端的使用 Servlet 对象的 Java 方法，如 HttpServlet Request、HttpServletResponse、HttpSession 等。

18.5　日　　志

18.5.1　Log4j——最流行的日志工具

Log4j 是 Apache 的一个开放源代码项目，通过使用 Log4j，可以控制日志信息输送的目的地是控制台、文件、GUI 组件，甚至是套接口服务器、NT 的事件记录器、UNIX Syslog 守护进程等；可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，能够更加细致地控制日志的生成过程。最重要的是这些可以通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。

18.5.2　Jakarta Commons Logging——Jakarta 的优秀日志工具

Jakarta Commons Logging（JCL）提供的是一个日志（Log）接口（Interface），同时兼顾轻量级和不依赖于具体的日志实现工具。它提供给中间件/日志工具开发者一个简单的日志操作抽象，允许程序开发人员使用不同的具体日志实现工具。用户被假定已熟悉某种日志实现工具的更高级别的细节。JCL 提供的接口对其他一些日志工具，包括 Log4j、Avalon LogKit 和 JDK 1.4 等，进行了简单的包装，此接口更接近于 Log4j 和 LogKit 的实现。

18.6　小　　结

Web 应用的设计是 Web 开发一个很重要的环节，如果设计不好，就会前功尽弃，而现在的 Web 开发也变得负责起来，对于开发时选择什么样的框架结构，使用什么样的测试工具，如何记录日志信息等都是需要事先考虑清楚的，在本章中介绍一些常用的相关工具，在这里只是简单的介绍，具体如何使用在后面的章节都会有详细的介绍。

第 19 章　MVC 模式实现——Struts

Struts 是 MVC 的一种实现，它很好地结合了 JSP、Java Servlet、JavaBeans、Taglib 等技术。Struts 已经成为了用 Java 创建 Web 应用的一个最流行的框架工具，Struts 所实现的 MVC 模式给 Web 应用带来了良好的层次划分，同时也提供了一系列的工具来简化 Web 应用的开发，成为基于 MVC 模式的 Web 应用最经典的框架。

19.1　快速体验 Struts——使用 Struts 框架的简单应用实例

19.1.1　建立 Struts 开发环境

要使用 Struts 创建 Web 应用，需要下载安装支持 Struts 的 JAR 文件，并把它放到 Web 应用的 lib 目录下，这些文件可以从其官方网站 http://struts.apache.org/上免费下载，最新版本是 1.2.7，下载其二进制版本 struts-1.2.7.zip 后，按照如下步骤建立发布第一个 Struts 应用。

（1） 将 struts-1.2.7.zip 解压缩到本地硬盘，设定解压后的目录为<Strust_HOME>。

（2）把<Strust_HOME>\webapps\struts-examples.war 文件复制到<TOMCAT_HOME>\webapps 目录下。

（3）重新启动 Tomcat，在浏览器地址栏中输入如下地址：http://localhost:8080/struts-examples/。可以看到页面显示如图 19.1 所示。

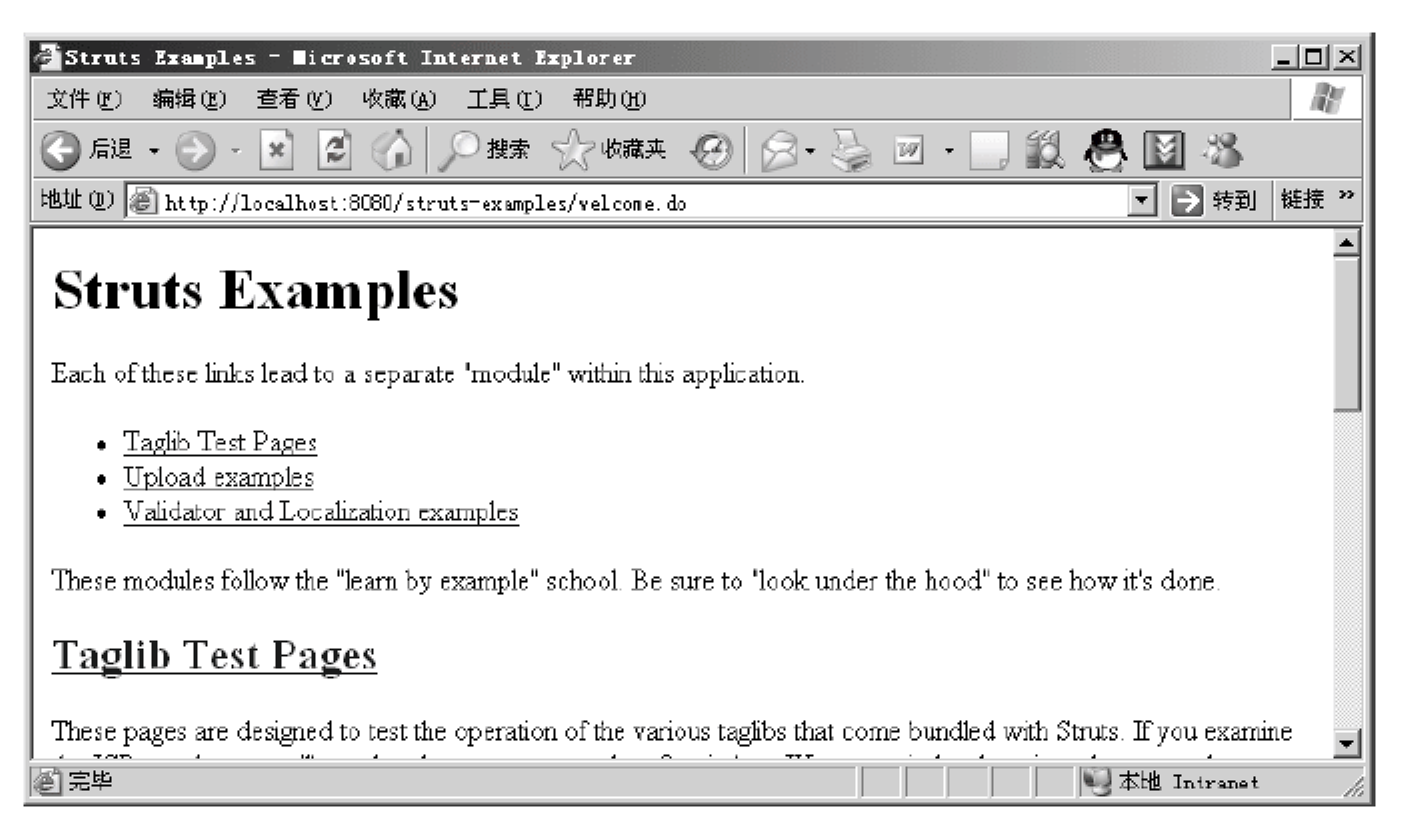


图 19.1　发布第一个 Struts 应用

看起来，Struts 的应用还是很容易发布的。其实要使用 Struts 创建 Web 应用关键需要<Strust_HOME>\lib 目录下的如下 JAR 文件的支持：

- antlr.jar

- commons-beanutils.jar
- commons-digester.jar
- commons-validator.jar
- jakarta-oro.jar
- struts.jar

这些 JAR 文件是编译 ActionServlet、Action 等 Struts 关键组建的必需包，所以需要把这些文件加入到 classpath 中，用于编译 Struts 应用的 Java 类文件，同时，在发布应用时，还应该把这个文件复制到该应用的 WEB-INF\lib 目录下，以为 Web 应用程序的运行提供支持。

19.1.2　实例介绍

在本小节介绍一个简单的例子，在这个例子中，可以通过超链接和表单提交的方式访问在配置文件中配置的视图。

使用表单提交的方式访问视图的流程如图 19.2 所示。

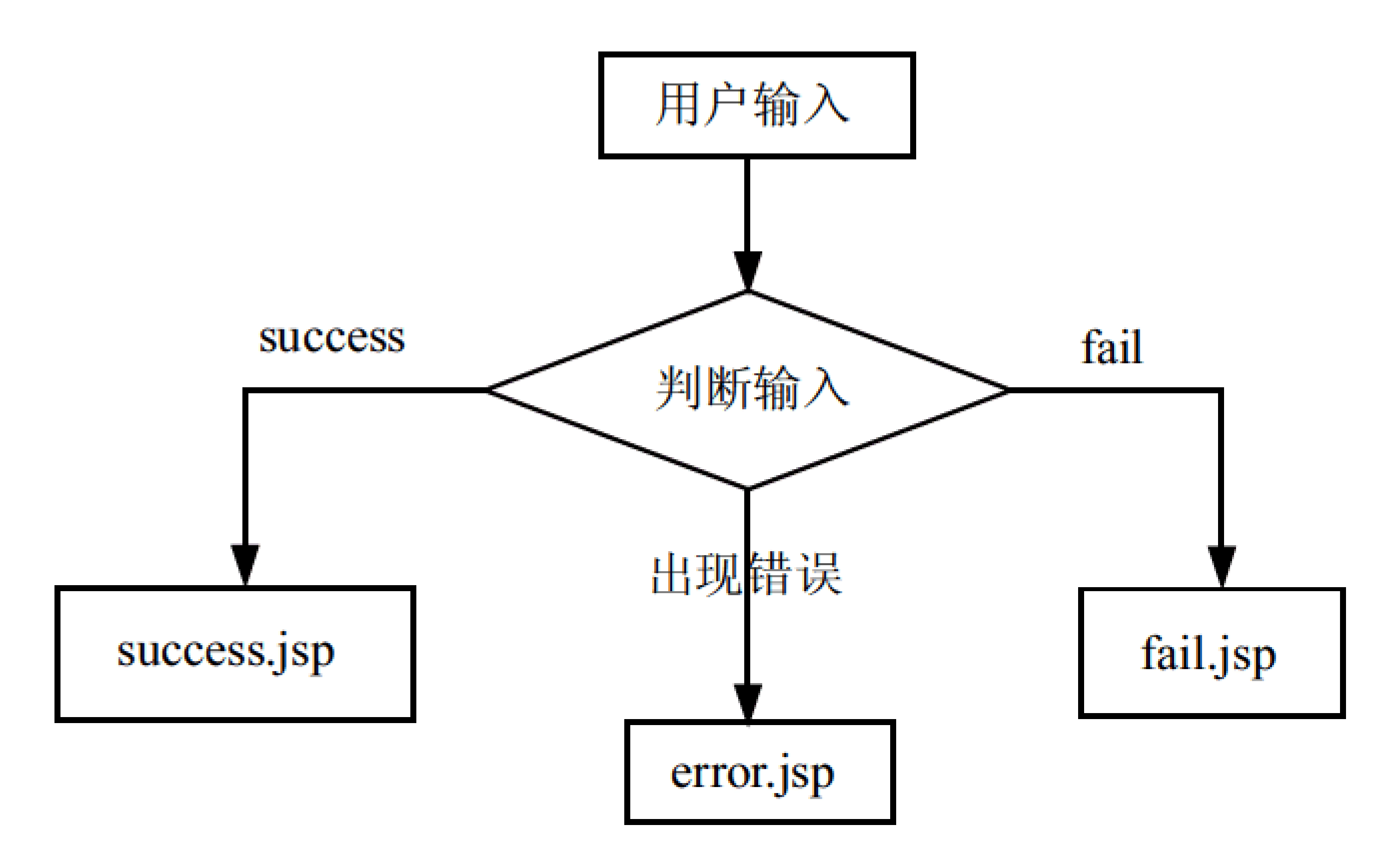


图 19.2　使用表单提交的方式访问视图的流程

使用超链接的方式访问视图就比较简单了，与普通的超链接是一样的，这里就不介绍了。

19.1.3　创建 Model 组件

Model 就是在对用户请求的整个控制过程中，真正处理用户请求并保存处理结果的对象，在整个过程中，一般用 JavaBeans 把一些信息保存起来以便在各个对象之间传递。因为在 MVC 框架中，Model 对象是真正处理商业逻辑功能的对象，因此也就是框架中应用需求实现相关性最大的部分。

在 Struts 的实现里，Model 的具体表现形式就是 ActionForm 对象和与其对应的 Action 对象了。对用户提交表单的数据进行校验，甚至对数据进行预处理都能在 ActionForm 中完成。通常的应用中，一般是一个 Model 对象和一个请求页面对应的关系，但也可以一个 Model 对象对应多个页面请求。如果 struts-config.xml 配置文件没有指定一个 Model 对象对应的 Action，那么控制器将直接把（通过 Model 对象完成数据封装的）请求转到一个 View 对象。

在这个例子中，为了简化，没有使用 ActionForm 对象的验证方法，开发者可以在 ActionForm 的 validate 方法中对用户的输入进行验证，用户如果要尝试如何在 ActionForm 中对自己的数据进行验证，可以自己添加这个方法。完整的代码（GetParameterForm.java）如下：

```
package cn.ac.ict;
import org.apache.struts.action.ActionForm;

public class GetParameterForm extends ActionForm{
//一个属性值
	private String valu="null";

	public GetParameterForm() {
		}
//设置和获取属性值的方法
	public void setValu(String s) {
		valu = s;
		}
	public String getValu() {
		return valu;
		}
		}
```

除了 ActionForm 外，还需要一个 Action 对象，像上面说的那样，Action 是和应用需求实现相关性很大的，它真正处理商业逻辑功能，代码（ShowAction.java）如下：

```
package cn.ac.ict;

import java.lang.reflect.InvocationTargetException;
import java.util.Locale;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.*;
import org.apache.struts.util.*;

public final class ShowAction extends Action{
//定义 execute 方法
	public ActionForward execute(ActionMapping mapping,ActionForm form,
				HttpServletRequest request,HttpServletResponse response)
					throws Exception {
					Locale locale = getLocale(request);
					MessageResources messages = getResources(request);
					HttpSession session = request.getSession();
					GetParameterForm userform = (GetParameterForm) form;
//判断用户的输入信息，并转发到不同的页面
						if(userform.getValu().equals("success")) {
					return(mapping.findForward("success"));
```

```
                    }else {
                        System.out.println(userform.getValu());
                        return(mapping.findForward("fail"));
                        }
                    }
                }
```

在这个类中只有一个方法 execute，它获得用户的请求信息，得到用户输入的字符串，并判断是 success 还是 fail，如果是 success 就转发到 success 视图（在配置文件中配置，对应 success.jsp），如果是 fail 就转发到 fail 视图（在配置文件中配置，对应 fail.jsp）。

19.1.4　创建 View 组件

根据上面的实例介绍，可以了解到在本实例中涉及 4 个 JSP 文件（还需要一个提供用户输入的页面），分别是 index.jsp、success.jsp、fail.jsp 和 error.jsp，在这里它们都是非常简单的，只是简单地输出一个字符串，它们的代码如下：

1．编写 index.jsp 文件

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ page import="java.util.*,java.sql.*,java.text.*,java.io.*"%>

<form name="form1" method="post" action="showinput.do">
输入 success 将返回到"success"页面，否则返回到"fail"页面<br><br>
input:<input type="text" name="valu">    <input type="submit" value="submit">
</form>
<br>
<a href="success.do">success</a><br>
<a href="fail.do">fail</a>
```

2．编写 success.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ page import="java.util.*,java.sql.*,java.text.*,java.io.*"%>

success!
```

3．编写 fail.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ page import="java.util.*,java.sql.*,java.text.*,java.io.*"%>

fail!
```

4．编写 error.jsp

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ page import="java.util.*,java.sql.*,java.text.*,java.io.*"%>

error page!
```

19.1.5　编写配置文件

Struts 的配置文件定义了全局转发、ActionMapping 类、ActionForm Bean 和 JDBC 数据源 4 个部分，在本实例中只需要配置前 3 种就可以了，下面是配置文件的完整代码：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
            "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
            "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
<!-- ==========  定义 Form Beans ==================================== -->
        <form-beans>
                <form-bean name="GetParameterForm" type="cn.ac.ict.GetParameterForm"/>
        </form-beans>

<!-- ==========定义全局转发============================== -->
        <global-forwards>
                <forward name="success" path="/success.do"/>
                <forward name="fail" path="/fail.do"/>
        </global-forwards>

<!-- ==========  定义 Action 映射  ================================ -->
        <action-mappings>
        <action
           path="/success"
           type="org.apache.struts.actions.ForwardAction"
           parameter="/success.jsp"/>
        <action
           path="/fail"
           type="org.apache.struts.actions.ForwardAction"
           parameter="/fail.jsp"/>
        <action path="/showinput"
                         type="cn.ac.ict.ShowAction"
                         name="GetParameterForm"
                         scope="request"
                         input="/index.jsp" >
                         <forward
              name="success"
              path="/success.jsp"/>
                         <forward
              name="fail"
              path="/fail.jsp"/>
    </action>
        </action-mappings>

</struts-config>
```

具体元素的意义将在后面的章节介绍。

这是 Struts 运行必需的一个配置文件，但除此之外，还需要在 web.xml 文件中配置使用控制器组件，一般控制器组建是 Struts 包中提供的 org.apache.struts.action.ActionServlet，web.xml 文件的完整代码如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <display-name>Simple Struts Application</display-name>
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
   </servlet>

<!-- Action Servlet 映射 -->
     <servlet-mapping>
          <servlet-name>action</servlet-name>
          <url-pattern>*.do</url-pattern>
     </servlet-mapping>

<!-- 欢迎文件列表 -->
     <welcome-file-list>
          <welcome-file>index.jsp</welcome-file>
     </welcome-file-list>

</web-app>
```

注意： 配置文件中不能包含中文注释，否则是无法识别的。

19.1.6　发布运行 Web 应用

上述文件都准备好后，可以按照如下步骤发布 Web 应用：

（1）编译 ActionForm 和 Action 的实现类，编译时，需要把 Struts 的支持包加入到类路径中，把编译后的字节码文件复制到 Web 应用的 WEB-INF\classes 目录下，存放的目录层次和包的层次是一致的，此时 Web 应用的目录结构如图 19.3 所示。

（2）启动 Tomcat，在浏览器地址栏中输入如下地址：http://localhost:8080/19/index.jsp，可以看到页面显示如图 19.4 所示。

当在页面中输入 success 或者单击下面的 success 链接时就会转向 success.do 页面，也就是 success.jsp，页面显示如图 19.5 所示。

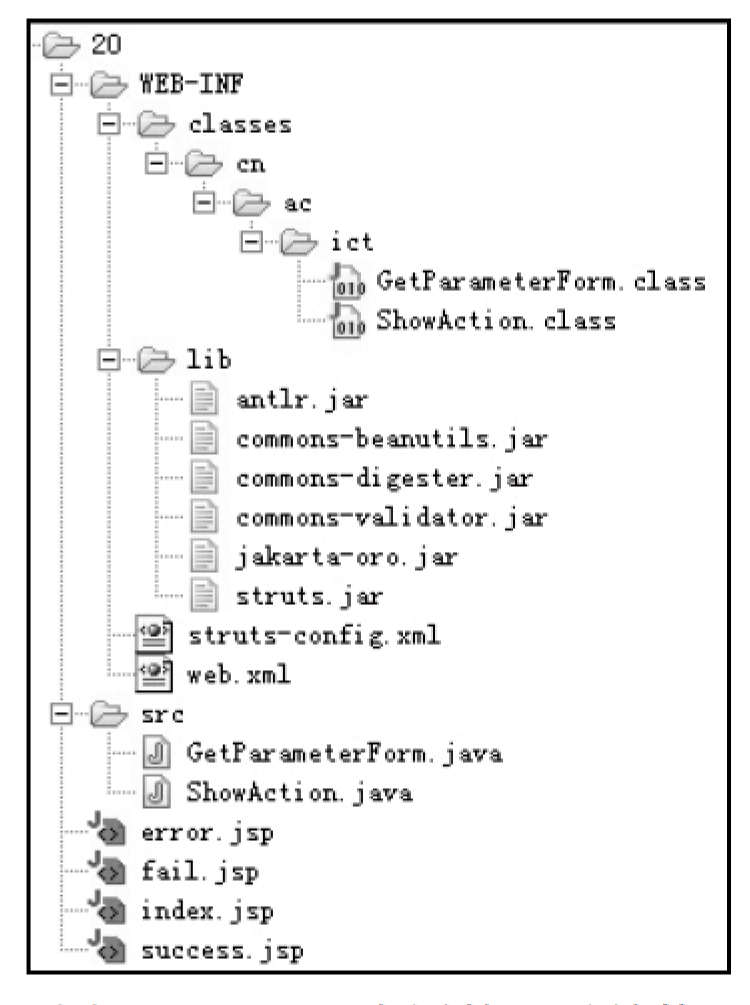


图 19.3　Web 应用的目录结构

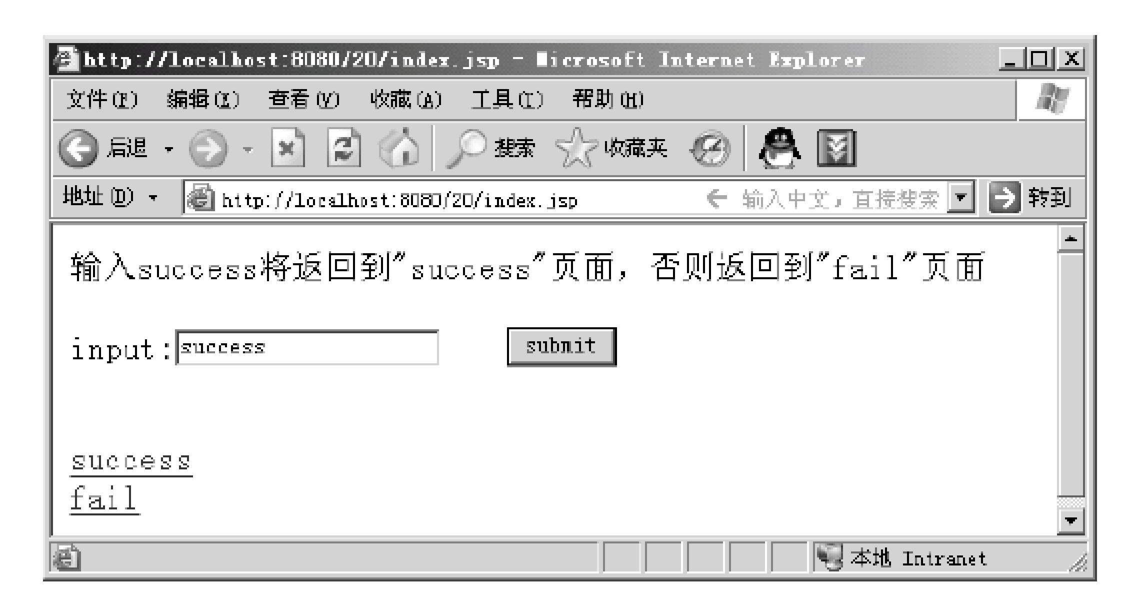


图 19.4　用户输入界面



图 19.5　success.do 页面

19.2　Struts 框架的体系结构与运行原理

Struts 是 Apache Jakarta 项目的组成部分。该项目的目标是为建立 Java Web 应用程序而提供的一个开源框架，目前一般使用的版本为 1.1，最新版本是 1.2。通过使用 Struts 框架可以改进和提高 Java Server Pages（JSP）、Servlet、标签库以及面向对象的技术在 Web 应用程序中的应用。应用 Struts 框架可以减少应用 MVC（Model-View-Controller）设计模式的开发时间，从而提高开发效率。

Struts 是 MVC 的一种实现，它很好地结合了 Jsp、Java Servlet、JavaBeans 和 Taglib 等技术。下面从 MVC 角度谈谈 Struts 框架体系结构和工作原理，并解释一下 Struts 配置文件，用于使用 Struts 开发 Web 应用。

19.2.1　从 Struts 的组件来看 Struts 的工作原理

如图 19.6 所示是 Struts 框架的体系结构。

（1）控制器：在 Struts 中，ActionServlet 起着一个控制器（Controller）的作用。ActionServlet 是一个通用的控制组件。这个控制组件提供了处理所有发送到 Struts 的 HTTP 请求的入口点。它截取和分发这些请求到相应的动作类(这些动作类都是 Action 类的子类)。另外控制组件也负责用相应的请求参数填充 ActionForm（通常称之为 FormBean），并传给

动作类（通常称之为 ActionBean）。动作类实现核心商业逻辑，它可以访问 JavaBeans 或调用 EJB。所有这些控制逻辑利用 Struts-config.xml 文件来配置。

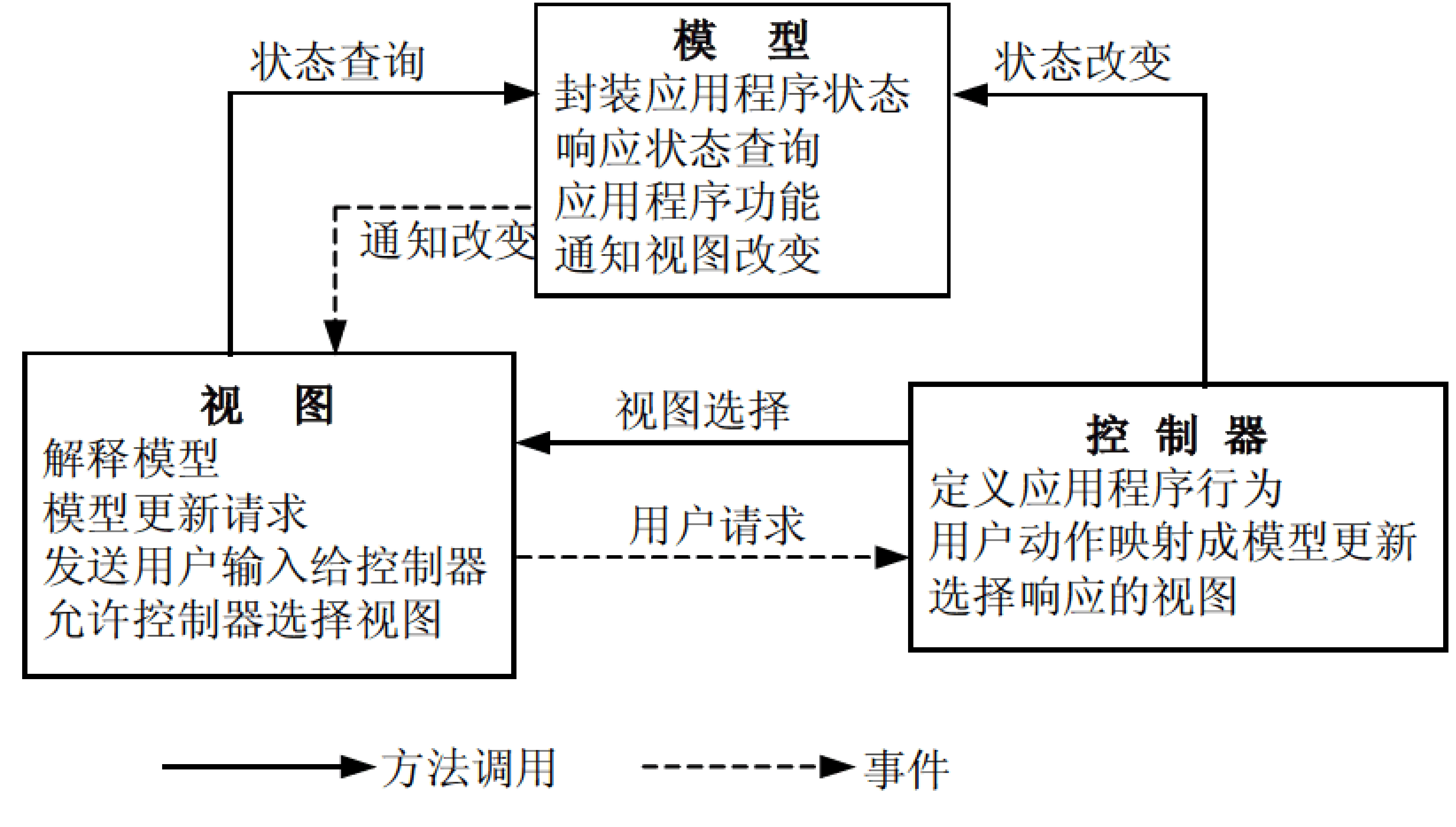


图 19.6　Struts 的体系结构

（2）视图：主要是由 JSP 来控制页面输出的。它接收到 ActionForm 中的数据，利用 HTML、Taglib、Bean、Logic 等显示数据。

（3）模型：在 Struts 中，主要存在 3 种 Bean，分别是：Action、ActionForm、EJB 或者 JavaBeans。ActionForm 用来封装客户请求信息，Action 取得 ActionForm 中的数据，再由 EJB 或者 JavaBeans 进行处理。

如图 19.7 所示是 Struts 体系结构中的组件图，下面就结合图 19.6 讲解 Struts 的具体构成。

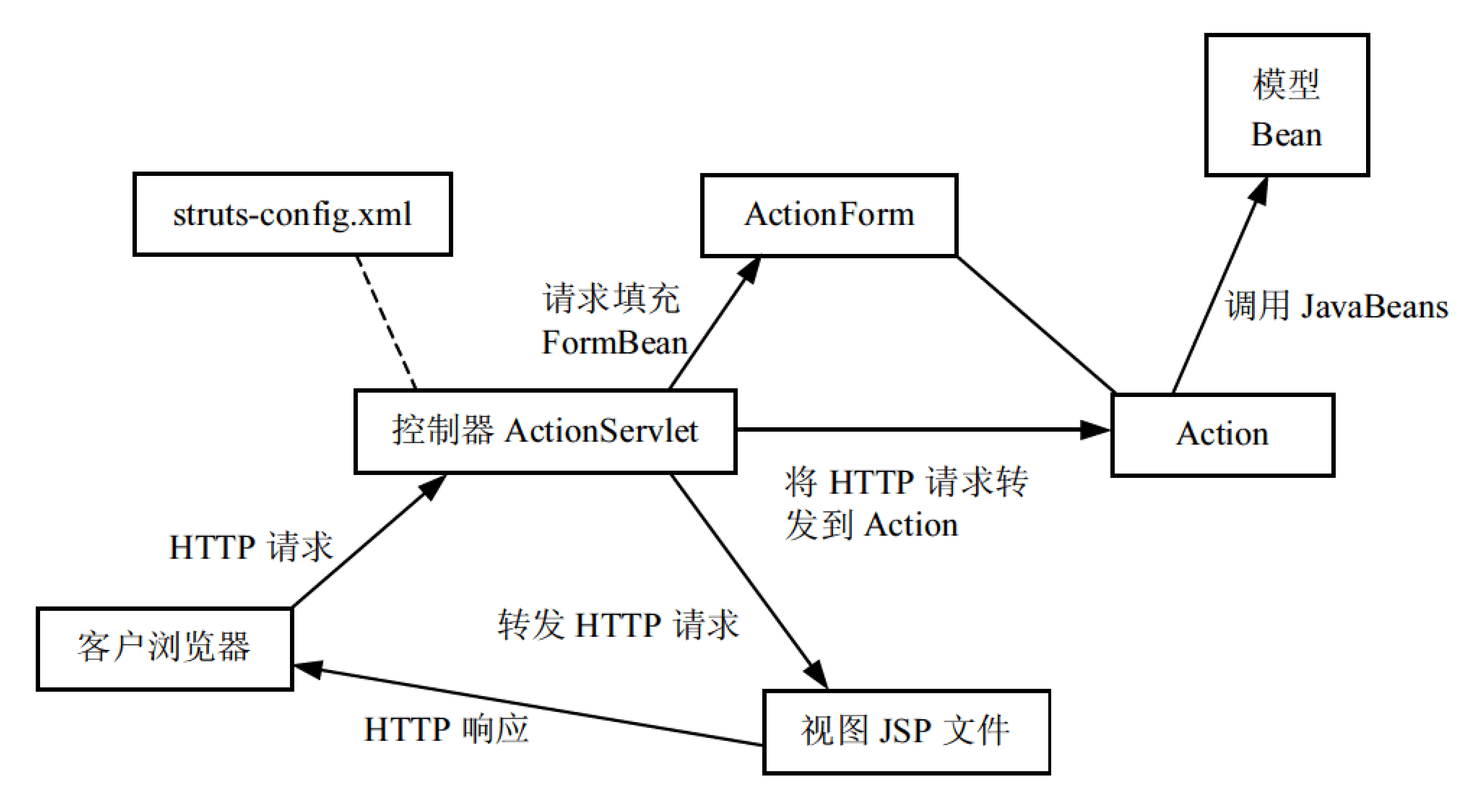


图 19.7　Struts 体系结构中的组件

图 19.7 显示了 ActionServlet（Controller）、ActionForm（Form State）和 Action（Model Wrapper）之间的最简关系。体系结构中所使用的组件如表 19.1 所示。

表 19.1　Struts体系结构中所使用的组件

组 件 名 称	组件所起的作用
ActionServlet	控制器
ActionClass	包含事务逻辑
ActionForm	显示模块数据
ActionMapping	帮助控制器将请求映射到操作
ActionForward	用来指示操作转移的对象
ActionError	用来存储和回收错误
Struts标记库	可以减轻开发显示层次的工作

19.2.2　Struts 配置文件 struts-config.xml

struts-config.xml 是 Struts 的配置文件，文件的配置包括全局转发、ActionMapping 类、ActionForm Bean 和 JDBC 数据源 4 个部分。

1．配置全局转发

全局转发用来在 JSP 页面之间创建逻辑名称映射。转发都可以通过对调用操作映射的实例来获得，例如：

```
actionMappingInstace.findForward("logicalName");
```

下面是全局转发的例子：

```
<global-forwards>
//其中全局转发的名字是 bookCreated，转发的目标路径是/BookView.jsp
        <forward name="bookCreated" path="/BookView.jsp"/>
</global-forwards>
```

2．配置 ActionMapping

ActionMapping 对象帮助进行框架内部的流程控制，它们可将请求 URI 映射到 Action 类，并且将 Action 类与 ActionForm Bean 相关联。ActionServlet 在内部使用这些映射，并将控制转移到特定 Action 类的实例。所有 Action 类使用 perform()方法实现特定应用程序代码，返回一个 ActionForward 对象，其中包括响应转发的目标资源名称。ActionMapping 的属性如表 19.2 所示。下面是一个例子：

```
<action-mappings>
<action path="/createBook" type="BookAction" name="bookForm" scope="request" input= "/Create
Book.jsp">
</action>
<forward name="failure" path="/CreateBook.jsp"/>
<forward name="cancel" path="/index.jsp"/>
</action-mappings>
```

表 19.2　ActionMapping的属性

属　　性	描　　述
path	Action类的相对路径
name	与本操作关联的Action Bean的名称
type	连接到本映射的Action类的全称（要有包名）
scope	ActionForm Bean的作用域（请求或会话）
prefix	用来匹配请求参数与Bean属性的前缀
suffix	用来匹配请求参数与Bean属性的后缀
attribute	作用域名称
className	ActionMapping对象的类的完全限定名，默认的类是org.apache.struts.action.ActionMapping
input	输入表单的路径，指向Bean发生输入错误必须返回的控制
unknown	设为true，操作将被作为所有没有定义的ActionMapping的URI的默认操作
validate	设置为true，则在调用Action对象上的perform()方法前，ActionServlet将调用ActionForm Bean的validate()方法来进行输入检查

通过<forward>元素可以定义资源的逻辑名称，该资源是 Action 类的响应要转发的目标，forward 属性如表 19.3 所示。

表 19.3　forward属性

属　　性	描　　述
id	ID
className	ActionForward类的完全限定名，默认是org.apache.struts.action.ActionForward
name	操作类访问ActionForward时所用的逻辑名
path	响应转发的目标资源的路径
redirect	若设置为true，则ActionServlet使用sendRedirect()方法来转发资源

3．配置 ActionForm Bean

ActionServlet 使用 ActionForm 来保存请求的参数，这些 Bean 的属性名称与 HTTP 请求参数中的名称相对应，控制器将请求参数传递到 ActionForm Bean 的实例，然后将这个实例传送到 Action 类。ActionForm Bean 的属性如表 19.4 所示。下面是一个配置 ActionForm Bean 的例子：

```
<form-beans>
<form-bean name="bookForm" type="BookForm"/>
</form-beans>
```

表 19.4　配置ActionForm Bean的属性

属　　性	描　　述
id	ID
className	ActionForm Bean的完全限定名，默认值是org.apache.struts.action.ActionFormBean
name	表单Bean在相关作用域的名称，这个属性用来将Bean与ActionMapping进行关联
type	类的完全限定名

4．配置 JDBC 数据源

用<data-sources>元素可以定义多个数据源，其所具有的属性如表 19.5 所示。

表 19.5　data-sources的属性

属　　性	描　　述
id	ID
key	Action类使用这个名称来寻找连接
type	实现JDBC接口的类的名称

这些属性还不足以定义一个完整的数据源，其他的属性需要子元素<set-property>定义（set-property 的属性如表 19.6 所示），在 Struts 1.1 版本中已不再使用，但仍可用<data-source>元素。例如：

```
<data-sources>
<data-source id="DS1" key="conPool" type="org.apache.struts.util.GenericDataSource"
<set-property id="SP1" autoCommit="true" description="Example Data Source Configuration"
driverClass="org.test.mm.mysql.Driver" maxCount="4"
minCount="2" url="jdbc:mysql://localhost/test" user="struts" password="ghq123" />
<data-source/>
</data-sources>
```

表 19.6　set-property的属性

属　　性	描　　述
description	数据源的描述
autoCommit	数据源创建的连接所使用的默认自动更新数据库模式
driverClass	数据源所使用的类，用来显示JDBC驱动程序接口
loginTimeout	数据库登录时间的限制，以秒为单位
maxCount	最多能建立的连接数目
minCount	要创建的最少连接数目
password	数据库密码
readOnly	创建只读的连接
user	访问数据库的用户名
url	JDBC的URL

通过指定关键字名称，Action 类可以访问数据源，例如：

```
javax.sql.DataSource ds = servlet.findDataSource("conPool");
javax.sql.Connection con = ds.getConnection();
```

19.3　Struts 组件介绍

19.3.1　使用 ActionServlet 控制器类分发请求

Struts 控制器组件负责接收用户的请求、更新模型以及选择合适的视图组件返回给客户

端。控制器组件有助于将模型和视图分离，有了这种分离可以在同一个模型的基础上得心应手地开发多种模型的视图。

Struts 控制器组件主要包括 ActionServlet 框架中央控制器、RequestProcessor 每个子应用的模块处理器，ActionServlet 负责处理具体的业务的组件。Struts 采用 ActionServlet 和 RequestProcessor 组件进行集中控制，并且采用 Action 组件来完成具体的业务单项处理。控制器组件的控制机制主要是接受用户请求，根据用户的请求调用合适的模型来执行业务逻辑，获取业务逻辑的结果，根据当前状态以及业务逻辑执行结果选择合适的视图组件返回给客户端。

ActionServlet 继承自 javax.servlet.http.HttpServlet 类，其在 Struts 中扮演的角色是中心控制器。它提供一个中心位置来处理全部的终端请求。控制器 ActionServlet 主要负责将 HTTP 的客户请求信息组装后，根据配置文件的指定描述，转发到适当的处理器。

按照 Servelt 的标准，所有的 Servlet 必须在 Web 配置文件（web.xml）声明。同样，ActoinServlet 必须在 Web 应用的配置文件（web.xml）中描述，有关配置信息如下：

```
<servlet>
<servlet-name>action</servlet-name>
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
```

全部的请求 URI 以*.do 的模式存在并映射到这个 Servlet，其配置如下：

```
<servlet-mapping>
<servlet-name>action</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
```

一个该模式的请求 URI 符合如下格式：

```
http://www.my_site_name.com/mycontext/actionName.do
```

中心控制器为所有的表示层请求提供了一个集中的访问点。这个控制器提供的抽象概念减轻了开发者建立公共应用系统服务的困难，如管理视图、会话及表单数据。它也提供一个通用机制如错误及异常处理、导航、国际化、数据验证和数据转换等。

当用户向服务器端提交请求时，实际上信息是首先发送到控制器 ActionServlet，一旦控制器获得了请求，其就会将请求信息转交给一些辅助类（Help Classes）处理。这些辅助类知道如何去处理与请求信息所对应的业务操作。在 Struts 中，这个辅助类就是 org.apache.struts.action.Action。通常开发者需要自己继承 Aciton 类，从而实现自己的 Action 实例。

19.3.2　使用 Action 组件分离请求和业务

ActionServlet 全部提交的请求都被控制器委托到 RequestProcessor 对象。RequestProcessor 使用 struts-config.xml 文件检查请求 URI 找到动作 Action 标识符。

一个 Action 类的角色就像客户请求动作和业务逻辑处理之间的一个适配器（Adaptor），其功能就是将请求与业务逻辑分开。这样的分离使得客户请求和 Action 类之间可以有多个

点对点的映射。而且 Action 类通常还提供了其他的辅助功能，比如：认证（Authorization）、日志（Logging）和数据验证（Validation）。

Action 最为常用的是 execute()方法。

```
public ActionForward execute(ActionMapping mapping,ActionForm form,
                                    javax.servlet.ServletRequest request,
                                    javax.servlet.ServletResponse response)
                          throws java.io.IOException,javax.servlet.ServletException
```

注意： 以前的 perform 方法在 Struts 1.1 中已经不再支持。

当 Controller 收到客户的请求时，在将请求转移到一个 Action 实例时，如果这个实例不存在，控制器会首先创建它，然后会调用这个 Action 实例的 execute()方法。

Struts 为应用系统中的每一个 Action 类只创建一个实例。因为所有的用户都使用这一个实例，所以必须确定 Action 类运行在一个多线程的环境中。图 19.8 显示了一个 execute()方法如何被访问：

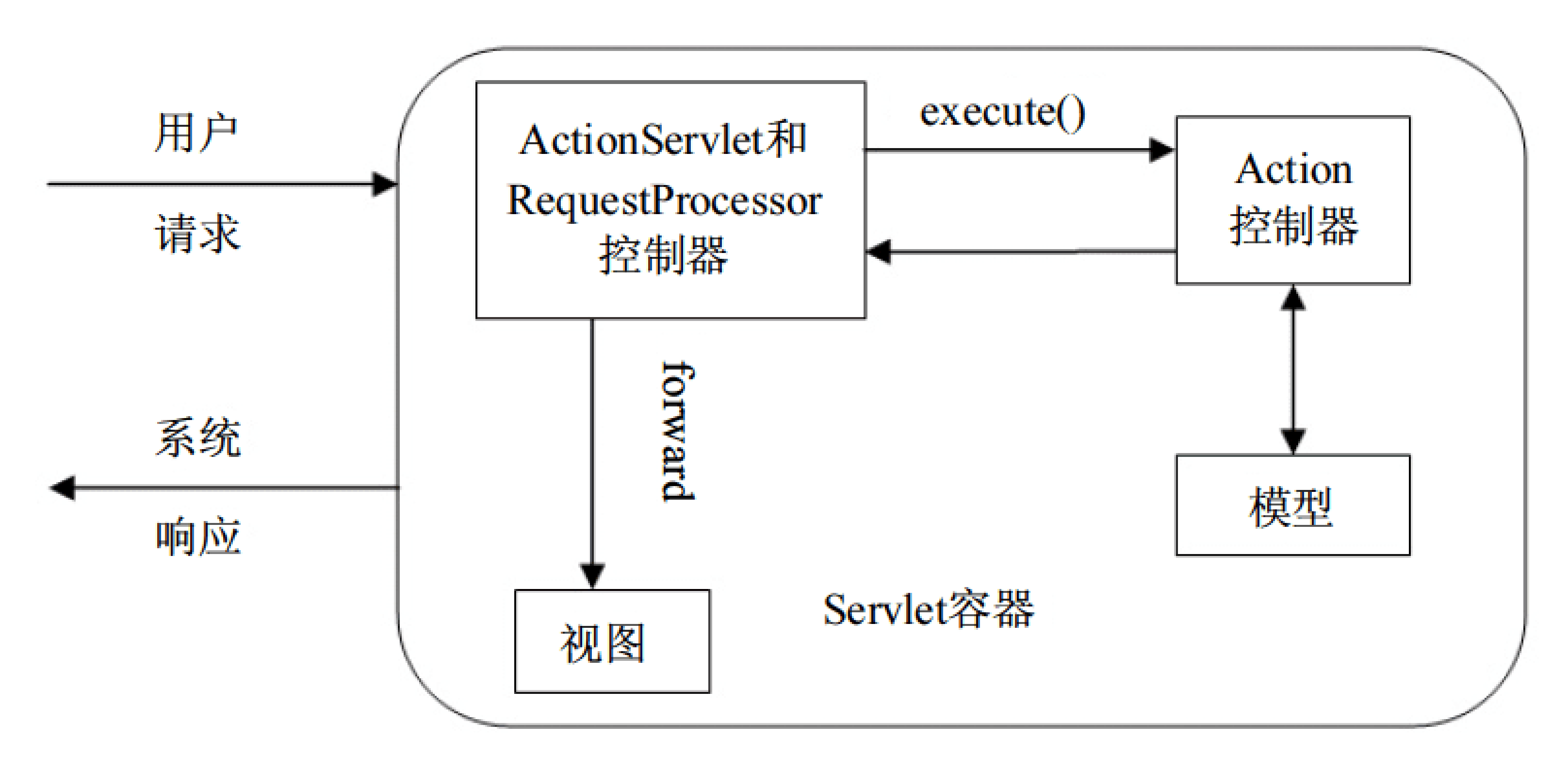


图 19.8　Action 实例的 execute()方法

注意： 开发者继承的 Action 子类，必须重写 execute()方法，因为 Action 类在默认情况下是返回 null 的。

19.3.3　使用 ActionMapping 映射 Action

上面讲到了一个客户请求是如何被控制器转发和处理的，但控制器如何知道什么样的信息转发到什么样的 Action 类呢？这就需要一些与动作和请求信息相对应的映射配置说明。

在 Struts 中，这些配置映射信息存储在特定的 XML 文件（比如 struts-config.xml）。这些配置信息在系统启动时被读入内存，供 Struts 在运行期间使用。在内存中，每一个<action>元素都与 org.apache.struts.action.ActionMapping 类的一个实例对应。下面就显示了一个登录的配置映射：

```
<action-mappings>
  <action    path="/logonAction"
             type="com.test.LogonAction"
             name="LogonForm"
             scope="request"
```

```
                input="logoncheck.jsp"
validate="false">
<forward name="welcome" path="/welcome.jsp"/>
<forward name="failure" path="/logon_failure.jsp "/>
</action>
</action-mappings>
```

上面的配置表示：当可以通过/logonAction.do（此处假设配置的控制器映射为*.do）提交请求信息时，控制器将信息委托 com.test.LogonAction 处理。调用 LogonAction 实例的 execute()方法。同时将 Mapping 实例和所对应的 LogonForm Bean 信息传入。其中 name= LogonForm，使用 form-bean 元素所声明的 ActionForm Bean。

19.3.4　ActionForm Bean 获取表单数据

在上面讲解 ActionServlet、ActionClasses 和 ActionMapping 时，都提到了 ActionForm Bean 的概念。一个应用系统的消息转移（或者说状态转移）的非持久性数据存储，是由 ActionForm Bean 负责保持的。

ActionForm 派生的对象用于保存请求对象的参数，因此它们和用户紧密联系。

一个 ActionForm 类被 RequestProcessor 建立。这是发生在已完成提交到一个 URL 后，该 URL 为映射到控制器 Servlet 而不是 JSP 和相应的动作映射指定的表单属性的。在这种情况下，如果没有在指定的活动范围内找到，RequestProcessor 将尝试寻找可能导致创建一个新 ActionForm 对象的表单 Bean。该 ActionForm 对象在指定的活动范围内被<action>元素的 name 属性找到。

RequestProcessor 随后将重新安排表单属性，用请求时参数填充表单，随即调用表单对象的 validate(…)方法以履行服务器端用户输入验证。仅当 ActionMapping 对象中 validate 属性被设为 true 时，validate(…)方法被调用，这就是默认的行为。

request.getParameterValues(parameterName)被用于得到一个 String[]对象，它用于表单填充；验证的结果应该是一个 ActionErrors 对象，用 org.apache.struts.taglib.html.ErrorsTag 来显示验证错误给用户。ActionForm 也可以被用于为当前用户保存即将被一个视图引用的中间模型状态。

当一个表单对象被 RequestProcessor 找到，它被传递到请求处理器的 execute(…)方法。一个 ActionForm 对象也可以被请求处理器建立。表单对象建立的目的是提供中间模型状态给使用请求范围 JSP；这将确保对象不会在有效性过期后仍然存在。默认情况下，所有表单都被保存为会话范围。会话中表单对象脱离有效性的存在可能导致浪费内存，同样地，请求处理器必须跟踪保存在会话中的表单对象的生命周期。一个好的捕获表单数据的实践是为横跨多用户交互的相关表单用一个单独的表单 Bean。表单 Bean 也可以在反馈时用来储存能够被自定义标签改变的中间模型状态。在视图中标签用法应避免结合 Java 代码，因此要有一个好的任务划分，Web 生产组主要处理标志，而应用开发组主要处理 Java 代码。标签因素退出访问中间模型状态的逻辑；当访问嵌套的对象或当通过聚集列举时这个逻辑可能很复杂。

注意：（1）在 Struts 1.1 中，ActionForm 的校验功能逐渐被剥离出来（当然依然可以使用），使用了 validator 对整个应用系统的表单数据验证进行统一管理。

（2）直接从 ActionFrom 类继承的 reset()和 validate()方法并不能实现什么处理功能，所以有必要自己重新覆盖。

19.4　使用 Struts 开发用户登录系统——Struts 实例

在本节中介绍一个用户登录的例子，在这个例子中，实现根据配置信息确定用户的合法性，如果用户的用户名和密码都是正确的，则让用户登录系统，而所用用户的信息是保存在一个资源文件中的，读者完全可以改为使用数据库保存用户信息。

19.4.1　开发模型组件

Struts 的 Model 组件包括 ActionForm Bean、系统状态 Beans 和商业逻辑 Beans，其中系统状态 Beans 和商业逻辑 Beans 通常表示为一组一个或多个的 JavaBeans 类。

在本实例中 Model 组件由 3 个类构成，其中一个是 LoginForm 类，一个是 LoginAction 类，最后一个是 LogoutAction 类，下面分别介绍这 3 个类以及它们的作用。

1．开发 LoginForm 类

LoginForm 类包含的属性和 Login.jsp 视图中表单的各个输入参数的名称一一对应，它扩展了 ActionForm 类。

在 LoginForm 类中定义了两个属性：username 和 password，以及它们对应的 setter 和 getter 方法，这些方法的定义是符合 JavaBeans 的定义规则的。下面是 LoginForm.java 的源代码：

```
package cn.ac.ict;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public class LoginForm extends ActionForm{
//这个 Bean 的属性
      private String password = null;
      private String username = null;

      // password 属性的获取和设置方法
      public String getPassword() {
       return (this.password);
       }
      public void setPassword(String password) {
```

```
        this.password = password;
        }

// username 属性的获取和设置方法
      public String getUsername() {
        return (this.username);
        }
      public void setUsername(String username) {
        this.username = username;
        }
      public void reset(ActionMapping mapping,HttpServletRequest request) {
        setPassword(null);
        setUsername(null);
        }
//用于验证用户输入是否合法
      public ActionErrors validate(ActionMapping mapping,HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if ((username == null) || (username.length() < 1))
             errors.add("username",new ActionError("error.username.required"));
        if ((password == null) || (password.length() < 1))
             errors.add("password",new ActionError("error.password.required"));
        return errors;
        }

}
```

在 LoginForm 类中除定义了符合 JavaBeans 规则的属性和方法外，还定义了一个 validate 方法，它是用来对客户提交的表单数据进行检查的方法。

在本实例应用中，用户名和密码都是不允许为空的，当任何一个为空，都会提示错误，而不同的项缺失也会提示不同的错误，如果用户名为空，就会抛出“error.username.required”错误，表示需要填写用户名。其中“error.username.required”只是一个错误信息的标志，当抛出这个错误时，系统会查找资源文件 application.properties（放在 WEB-INF\classes\resources 目录下），得到需要输出的错误信息是“Username is required”。同样，如果密码没有填写就会提示“Password is required”。

编译这个文件需要把 struts.jarJAR 文件加入到类路径中。

发布这个 LoginForm 类时，需要在 struts-config.xml 文件的<form-beans>元素中添加如下子元素：

```
<form-bean name="loginForm" type="cn.ac.ict.LoginForm" />
```

其中 name 属性为这个 ActionForm 指定了一个名字，供需要进行表单验证的资源参考（refference）时使用，type 定义了实现 ActionForm 的具体类的全名。

通过使用<form-beans>元素可以为 Web 应用定义多个 ActionForm，name 是每个 ActionForm 的惟一标识。

2．开发 LoginAction 类

LoginAction 类的作用是验证用户的合法性，并把用户的信息保存到 Session 中，并根据用户登录是否成功返回不同的信息。

客户（提交了表单）请求首先要经过 LoginForm 类，LoginForm 类验证通过后 ActionServlet 把请求转发给 LoginAction 类，LoginAction 类验证通过后，把请求转发给视图组件，其中成功和失败时转发到不同的视图组件，在 struts-config.xml 文件中定义如下：

```
<action
     path="/LoginSubmit"
     type="cn.ac.ict.LoginAction"
     name="loginForm"
     scope="request"
     validate="true"
     input="/pages/Login.jsp">
     <forward
          name="success"
          path="/pages/Welcome.jsp"/>
</action>
```

下面是 LoginAction 类的源代码：

```
package cn.ac.ict;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.*;

public class LoginAction extends Action {

//用于验证用户是否合法
      public boolean isUserLogin(String username,String password)
                  throws UserInfoException {

            return (UserInfo.getInstance().isValidPassword(username,password));
          // return true;
          }

          public ActionForward execute(ActionMapping mapping,
          ActionForm form,
          HttpServletRequest request,
          HttpServletResponse response)
          throws IOException, ServletException {
          //获取用户名和密码
          String username = ((LoginForm) form).getUsername();
          String password = ((LoginForm) form).getPassword();
```

```
        boolean validated = false;
        try {
          validated = isUserLogin(username,password);

        }catch (UserInfoException uie) {
//抛出错误信息
          ActionErrors errors = new ActionErrors();
          errors.add(ActionErrors.GLOBAL_ERROR,new ActionError("error.login.connect"));
          saveErrors(request,errors);
      // 返回输入页面
          return (new ActionForward(mapping.getInput()));
        }

        if (!validated) {
          //验证不通过时
          ActionErrors errors = new ActionErrors();
          errors.add(ActionErrors.GLOBAL_ERROR,new ActionError("error.login.invalid"));
          saveErrors(request,errors);
      // 返回输入页面
          return (new ActionForward(mapping.getInput()));
        }

        HttpSession session = request.getSession();
        session.setAttribute(Constants.USER_KEY, form);

        return (mapping.findForward(Constants.SUCCESS));

        }
}
```

在这个类中定义了两个方法：isUserLogin 和 execute。

- isUserLogin 方法用于验证某个用户的信息是否合法，用户信息的处理是通过辅助类 UserInfo 来实现的，它读取用户信息资源文件，并对用户的合法性进行验证。
- execute 方法是这个类的主要方法，它调用 isUserLogin 方法对客户身份进行验证，如果验证通过，会把客户的信息保存到 Session 中，并把请求转发到 Constants.SUCCESS 指定的视图页面：

```
        HttpSession session = request.getSession();
        session.setAttribute(Constants.USER_KEY, form);

        return (mapping.findForward(Constants.SUCCESS));
```

而如果验证失败，就会构造一个 ActionErrors 对象，并把客户请求转发到输入页面：

```
      ActionErrors errors = new ActionErrors();
      errors.add(ActionErrors.GLOBAL_ERROR,new ActionError("error.login.invalid"));
      saveErrors(request,errors);
      // 返回输入页面
      return (new ActionForward(mapping.getInput()));
```

编译这个文件需要把 struts.jarJAR 文件加入到类路径中。另外发布这个 LoginAction 类时，需要在 struts-config.xml 文件的<action-mappings>元素中添加如下子元素（元素属性的意义参考 19.2.3 节）：

```
<action
    path="/LoginSubmit"
    type="cn.ac.ict.LoginAction"
    name="loginForm"
    scope="request"
    validate="true"
    input="/pages/Login.jsp">
    <forward
        name="success"
        path="/pages/Welcome.jsp"/>
</action>
```

可以看到如果用户身份验证通过，请求将被转发到/pages/Welcome.jsp。

3．LogoutAction 类

LogoutAction 类的作用和 LoginAction 类是相对的，它完成的工作是将保存在 Session 中的用户删除，表示客户退出了登录，并把请求转发到合适的视图组件，下面是 LogoutAction 类的源代码：

```
package cn.ac.ict;

import java.io.IOException;
import java.util.Hashtable;
import java.util.Locale;
import java.util.Vector;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.*;
import org.apache.struts.util.MessageResources;

public class LogoutAction extends Action {

    public ActionForward execute(ActionMapping mapping,ActionForm form,HttpServletRequest
request,
HttpServletResponse response)throws IOException, ServletException {

        HttpSession session = request.getSession();
        LoginForm user = (LoginForm)session.getAttribute(Constants.USER_KEY);

        session.removeAttribute(Constants.USER_KEY);
        //将页面转发到成功页面
        return (mapping.findForward(Constants.SUCCESS));
    }
}
```

编译这个文件需要把 struts.jar 这个 JAR 文件加入到类路径中。

发布这个 LogoutAction 类时，需要在 struts-config.xml 文件的<action-mappings>元素中添加如下子元素（元素属性的意义参考 19.2.2 节）。

```
<action
    path="/Logout"
    type="cn.ac.ict.LogoutAction">
    <forward
        name="success"
        path="/pages/Welcome.jsp"/>
</action>
```

19.4.2　开发视图组件

Struts 的 View 组件包括一组 JSP 页面，这些 JSP 文件中可以包含 Struts 标签（有的元素可以不使用 Struts 标签表示，不过有的就必须使用 Struts 标签，而且 Struts 标签比较多，对于初学者来说有点难度）。在本实例中有 3 个 JSP 页面：Login.jsp、Welcome.jsp 和 index.jsp。

1．编写 index.jsp

index.jsp 页面是很简单的，它就是作为这个 Web 应用的默认页面。它什么都不显示，只是把请求转发给其他的页面，下面是 index.jsp 的源代码：

```
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<logic:forward name="welcome"/>
```

<logic:forward>标签用于转发请求。在这个页面中，使用了 Struts 标签，把请求转发给了 welcome。在 struts-config.xml 文件中定义了如下的全局转发：

```
<forward
    name="welcome"
    path="/Welcome.do"/>
```

在<action-mappings>元素中定义了如下元素：

```
<action
    path="/Welcome"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/pages/Welcome.jsp"/>
```

也就是说，根据 struts-config.xml 文件的定义，请求将最终被转发到 Welcome.jsp 页面。

2．编写 Welcome.jsp

Welcome.jsp 是 Web 应用的欢迎页面，其中提供了用于登录的链接，下面是 Welcome.jsp 的源代码：

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<html:html locale="true">
  <HEAD>
```

```
    <TITLE><bean:message key="welcome.title"/></TITLE>
  <html:base/>
  </HEAD>
<BODY>
  <logic:present name="user">
    <H3>Welcome <bean:write name="user" property="username"/>!</H3>
  </logic:present>

  <logic:notPresent scope="session" name="user">
    <H3><bean:message key="app.hello"/></H3>
  </logic:notPresent>

<html:errors/>
<UL>

<LI><html:link forward="login">Log in</html:link></LI>

  <logic:present name="user">
    <LI><html:link forward="logout">Log out</html:link></LI>
  </logic:present>
</UL>
</BODY>
</html:html>
```

可以看到这个文件中使用了大量的 Struts 标签，文件最开始就引入了 3 个标签文件，这些 Bean 文件中定义了大量的标签，Struts 标签库的描述如表 19.7 所示。

表 19.7　Struts标签库的描述

属　性	描　述
Bean Tags	Struts-bean 标签库中的JSP自定义标签的主要作用是用于访问Bean的属性，以及基于这些访问而定义新的Bean，使得页面可以通过脚本变量和页面内的属性来访问这些Bean
HTML Tags	Struts的HTML标签库包含用于生成Struts的用户界面的标签，该标签功能和HTML中的FORM相似，不同的是该标签将用户输入参数推入设定的ActionForm Bean当中
Logic Tags	Struts的logic标签库包含用于控制输出文本条件、对集合对象迭代以及应用流程控制方面的标签，该标签中的大部分标签功能在JSTL的core标签库中有相同或类似功能的标签
Nested Tags	Struts的nested标签库包含的标签库是扩展其他标签库的标签，该标签库使得Struts框架中的标签能够知道包含它的标签，以正确地设置其中的内嵌属性
Template Tags	该库包含的标签可用作页面创建动态的JSP模板，这些页面都拥有一个公共的外观或者共同的格式
Tiles Tags	Tiles框架提供一种模板机制，它能将布局和内容的职责分离

在这个文件的第 4 行使用了一个<html:html>标签，它来自于 struts-html 标签库，作用与 HTML 语言中的<html>元素一样，在 Struts 中不是必需的，完全可以使用 HTML 语言中的<html>元素代替。

在这个文件的第 6 行使用了一个<bean:message key="welcome.title"/>标签，它来自于 struts-bean 标签库，作用是从 Web 应用的资源配置文件 application.properties 中读取 key 指定的内容，例如在 application.properties 文件中包含如下内容：

```
welcome.title=Struts Login Example Application
```

则这个页面显示时，浏览器的标题栏将显示 Struts Login Example Application，如图 19.9 所示。

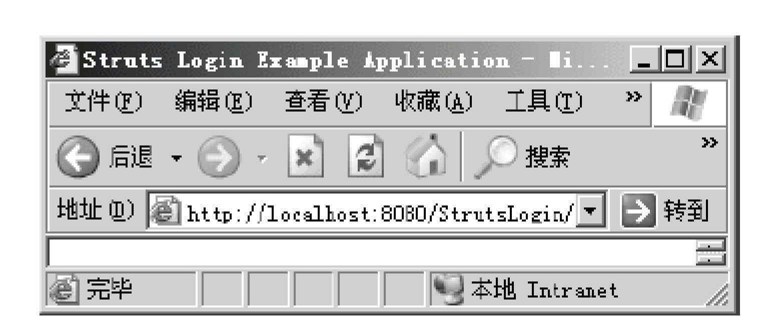


图 19.9　<bean:message>元素的作用

在这个文件的第 10 行～12 行使用了一个<logic:present>标签，它来自于 struts-logic 标签库，作用是检查 request 对象（由 scope 指定，如果不指定表示 request 范围）中参数（name 属性指定）是否存在，如果存在就执行这个元素之间的内容，也就是下面的内容：

```
<H3>Welcome <bean:write name="user" property="username"/>!</H3>
```

<logic:notPresent>标签的作用是和<logic:present>标签相反的，如果它所指定的属性不存在时，执行元素之间的内容：

```
<H3><bean:message key="app.hello"/></H3>
```

其中使用了<bean:message>标签，执行结果显示如下：

```
Welcome Guest Come Here!
```

3．编写 Login.jsp

Login.jsp 用于提供 FORM 表单，在这个页面中可以输入用户名和密码，其源代码如下：

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<HTML>
  <HEAD>
  <TITLE><bean:message key="welcome.title"/></TITLE>
  </HEAD>
<BODY>
<html:errors/>
<html:form action="/LoginSubmit" focus="username">
  <TABLE border="0" width="100%">
    <TR>
      <TH align="right"><bean:message key="app.username"/>:</TH>
      <TD align="left"><html:text property="username"/></TD>
  </TR>
  <TR>
    <TH align="right"><bean:message key="app.password"/>:</TH>
    <TD align="left"><html:password property="password"/></TD>
  </TR>
  <TR>
    <TD align="right"><html:submit/></TD>
    <TD align="left"><html:reset/></TD>
  </TR>
  </TABLE>
</html:form>
```

```
</BODY>
</HTML>
```

其中<html:text>、<html:password>、<html:submit/>和<html:reset/>分别相当于 HTML 语言中的<input type="text">、<input type="password">、<input type="submit">和<input type="reset">。

<html:form>元素用于定义表单，和 HTML 语言中的<form>作用相同。它的 Action 为 LoginSubmit，在 struts-config.xml 文件中将 LoginSubmit 定义为：

```
        <action
            path="/LoginSubmit"
            type="cn.ac.ict.LoginAction"
            name="loginForm"
            scope="request"
            validate="true"
            input="/pages/Login.jsp">
            <forward
                name="success"
                path="/pages/Welcome.jsp"/>
        </action>
```

19.4.3　开发辅助类

通过创建 Struts 的 Model 组件和 View 组件，可以发现在这个应用中使用了辅助类，包括：UserInfo.java、Constants.java 和 UserInfoException.java。

1．UserInfo.java

UserInfo.java 用于维护用户信息、获取用户名和密码、验证用户名和密码的正确性以及添加合法用户等，其源代码如下：

```
package cn.ac.ict;

import java.io.IOException;
import java.io.InputStream;
import java.io.FileOutputStream;
import java.util.Enumeration;
import java.util.Properties;

public class UserInfo{
     //存放用户信息的文件
    private static final String UserInfoFile =
                "resources/users.properties";
     //用户信息存放的格式
    private static final String UserInfoHeader =
                "${user}=${password}";

    private static UserInfo UserInfo = null;
```

```
      private static Properties p;
//构造函数被设置为 private，类外无法实例化这个类
      private UserInfo() throws UserInfoException {
       InputStream i = null;
       p = null;
    i = this.getClass().getClassLoader().getResourceAsStream(UserInfoFile);
       if (i==null) {
              throw new UserInfoException();
       }else {
              try {
                             p = new Properties();
                             p.load(i);
                             i.close();
                       }
                       catch (IOException e) {
                             p = null;
                             System.out.println(e.getMessage());
                             throw new UserInfoException();
                       }
                       finally {
                             i = null;
                       }
                 } // end else
      } // end UserInfo
//获取一个 UserInfo 的实例
      public static UserInfo getInstance() throws UserInfoException {
       if (null==UserInfo) {
              UserInfo = new UserInfo();
       }
       return UserInfo;
       }

      public String fixId(String userId) {
       return userId.toUpperCase();
       }
//判断是否为合法用户
      public boolean isValidPassword(String userId, String password) {

            if (password==null) return false;

            // 把用户名转化为大写字母
            String _userId = fixId(userId);

            // 用户不存在
            if (!isUserExist(_userId)) return false;

            // 用户存在并且密码正确
            return (password.equals(getPassword(_userId)));
            }
```

```
    public boolean isUserExist(String userId) {

        // 用户名为空时返回不存在
        if (userId==null) return false;
        // 用户不存在
        return !(p.getProperty(userId)==null);

    }
    //获得某个合法用户的密码
    public String getPassword(String userId)  {
          return p.getProperty(userId);
    }
     //取得所有合法的用户名
    public Enumeration getUserIds()  {
        return p.propertyNames();
    }

    public void setUser(String userId, String password) throws
            UserInfoException {
          if ((userId==null) || (password==null)) {
                throw new UserInfoException();
          }
        try {
        // 添加新的合法用户并存储
            p.put(fixId(userId), password);
            p.store(new FileOutputStream(UserInfoFile),
            UserInfoHeader);
        }catch (IOException e) {
          throw new UserInfoException();
        }
    }

}
```

在这个类的构造函数中，读取了用户信息文件，并把用户的信息保存到 properties 对象中，用于后面对用户的身份进行验证：

```
    private UserInfo() throws UserInfoException {
     InputStream i = null;
     p = null;
     i = this.getClass().getClassLoader().getResourceAsStream(UserInfoFile);
     if (null==i) {
            throw new UserInfoException();
     }else {
            try {
                        p = new Properties();
                        p.load(i);
                        i.close();
                  }
                  catch (IOException e) {
```

```
                    p = null;
                    System.out.println(e.getMessage());
                    throw new UserInfoException();
                }
                finally {
                    i = null;
                }
            } // end else
    }
```

而且这个类的构造函数被定义为 private，也就是说外部类是不可以实例化这个类的，因此这个类提供了一个获取 UserInfo 对象的方法 getInstance()，代码如下：

```
    public static UserInfo getInstance() throws UserInfoException {
          if (null==UserInfo) {
           UserInfo = new UserInfo();
     }
     return UserInfo;
     }
```

2．编写 Constants.java

Constants.java 定义了一些静态常量，供在其他地方引用，其源代码如下：

```
package cn.ac.ict;

public class Constants {

    /**
     * 应用程序的包名
     */
    public static final String Package = "app";

    /**
     * 存储在 Session 中的属性名称
     */
    public static final String USER_KEY = "user";

    /**
     *ActionForward 中的一个
     */
    public static final String SUCCESS = "success";

    /**
     * ActionForward 中的一个
     */
    public static final String LOGIN = "login";

    /**
     * ActionForward 中的一个
     */
```

```
    public static final String WELCOME = "welcome";

}
```

3．UserInfoException.java

UserInfoException.java 定义了一个异常，这个异常只继承了 Exception 类，不做任何事情，其代码如下：

```
package cn.ac.ict;
public class UserInfoException extends Exception {
    // 没有实现
}
```

19.4.4　创建配置文件

在编写好相关的类后，就可以编写 Struts 的配置文件了，下面是这个应用的 struts-config.xml 文件的完整代码：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

        <!-- 配置使用的 FormBean -->
   <form-beans>
     <form-bean name="loginForm" type="cn.ac.ict.LoginForm" />
   </form-beans>

    <global-exceptions>
    </global-exceptions>
        <!--    定义全局转发  -->
    <global-forwards>
        <forward
            name="welcome"
            path="/Welcome.do"/>
        <forward
            name="logout"
            path="/Logout.do"/>
        <forward
            name="login"
            path="/Login.do"/>

    </global-forwards>
<!-- ============ 定义 ActionMapping=========== -->
    <action-mappings>
            <!-- Default "Welcome" action -->
```

```
            <!-- Forwards to Welcome.jsp -->
        <action
            path="/Welcome"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/pages/Welcome.jsp"/>

        <action
            path="/Login"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/pages/Login.jsp"/>

        <action
            path="/LoginSubmit"
            type="cn.ac.ict.LoginAction"
            name="loginForm"
            scope="request"
            validate="true"
            input="/pages/Login.jsp">
            <forward
                name="success"
                path="/pages/Welcome.jsp"/>
        </action>

        <action
            path="/Logout"
            type="cn.ac.ict.LogoutAction">
            <forward
                name="success"
                path="/pages/Welcome.jsp"/>
        </action>
    </action-mappings>
        <!--    配置使用的 Message 资源    -->
    <message-resources parameter="resources.application"/>
</struts-config>
```

这个文件中各个元素的作用在 19.2.2 节中已介绍过，这里就不详细介绍了，值得注意的是这个文件的最后一个元素：

```
    <message-resources parameter="resources.application"/>
```

它指明了应用程序使用的消息资源文件，这个文件应该放在 Web 应用的 WEB-INF\classes\resources 目录下，其中保存了应用中使用的消息。

19.4.5　发布 Web 应用

在上一节中一步步创建了一个基于 Struts 的 Web 应用，下面介绍如何运行这个应用，读者可以按照如下步骤发布这个应用：

（1）编译所有的 Java 文件，这个过程需要把 Struts 的支持 JAR 文件放到类路径中，把编译后的字节码文件复制到 Web 应用的 WEB-INF\classes 目录下，存放的目录层次和包的层次是一致的，此时 Web 应用的目录结构如图 19.10 所示。

（2）启动 Tomcat，在浏览器地址栏中输入如下地址：http://localhost:8080/StrutsLogin/，可以看到页面如图 19.11 所示。

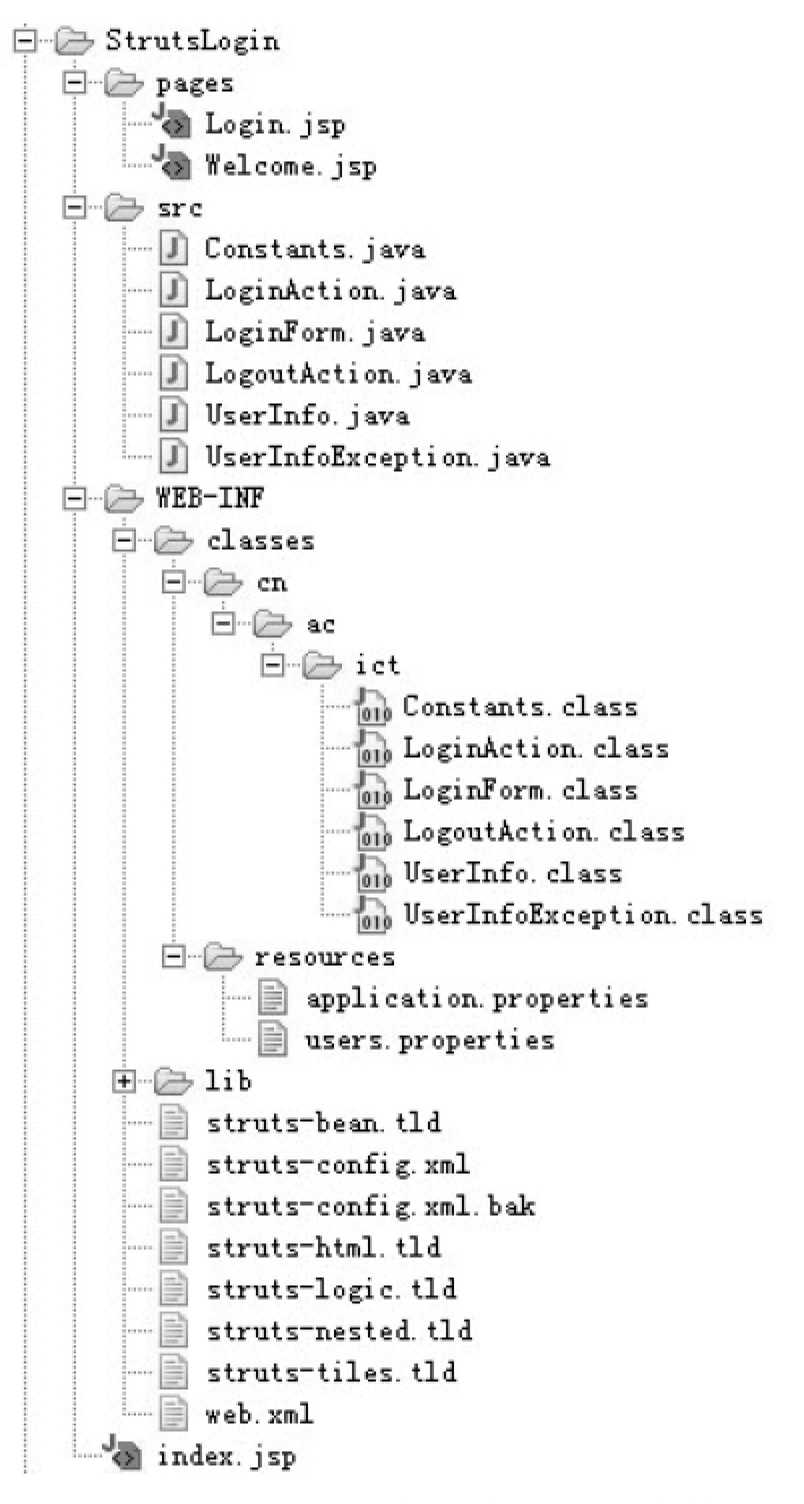


图 19.10　Web 应用的目录结构



图 19.11　Struts Login 应用

在地址栏中 http://localhost:8080/StrutsLogin/由于没有指定特点的资源，Tomcat 使用 Struts Login 应用的默认页面 index.jsp 响应客户。

在页面 index.jsp 中不做任何显示工作，只是把请求转发到 welcome 资源，根据 struts-config.xml 文件的定义 welcome 是全局转发，它对应了 Welcome.do 这个 Action，而在 struts-config.xml 文件中有如下定义：

```
<action
    path="/Welcome"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/pages/Welcome.jsp"/>
```

也就是说，index.jsp 页面最后把请求转发到了 Welcome.jsp，单击这个页面中的 Log in 链接后，链接到 Login.do 资源，根据 struts-config.xml 文件的如下定义：

```
<action
    path="/Login"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/pages/Login.jsp"/>
```

请求被转发到 Login.jsp 页面，所以页面显示如图 19.12 所示。



图 19.12　Login.jsp 页面

19.4.6　Web 应用分析

1．表单验证

用户可以在如图 19.12 所示的页面中输入信息并提交登录系统。客户在提交了请求后，服务器的执行流程如下：

（1）Tomcat 从 Web 应用的 web.xml 文件中对用户请求的 URL 进行匹配，发现了如下定义：

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

也就是说所有以.do 结尾的 URL 请求都会被命名为 Action 的 Servlet 进行处理。

（2）Tomcat 根据<servlet-name>找到了对 Action 的定义，它是 Struts 的 ActionServlet 类，也就是控制器类：

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
  </init-param>
  <init-param>
    <param-name>detail</param-name>
    <param-value>2</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
```

（3）ActionServlet 检查这个表单是否需要进行验证，它在 struts-config.xml 文件中发现路径为/LoginSubmit 的 Action 指定了使用名为 loginForm 的 ActionForm 进行验证，而 loginForm 在 struts-config.xml 文件中是这样定义的：

```
<form-beans>
  <form-bean name="loginForm" type="cn.ac.ict.LoginForm" />
</form-beans>
```

也就是说，当提交路径为 LoginSubmit.do 时，需要使用 cn.ac.ict.LoginForm 类进行验证，

这是控制器创建一个 cn.ac.ict.LoginForm 类的对象。

（4）控制器调用 cn.ac.ict.LoginForm 的 validate 方法，检查用户名和密码是否为空，如果有任何一个为空会返回一个 ActionErrors 的对象，然后控制器根据<action>元素的 input 属性把客户的请求转发给 Login.jsp，并显示错误信息。当只有用户名为空时弹出的错误消息，如图 19.13 所示。当用户名和密码都为空时弹出的错误消息，如图 19.14 所示。

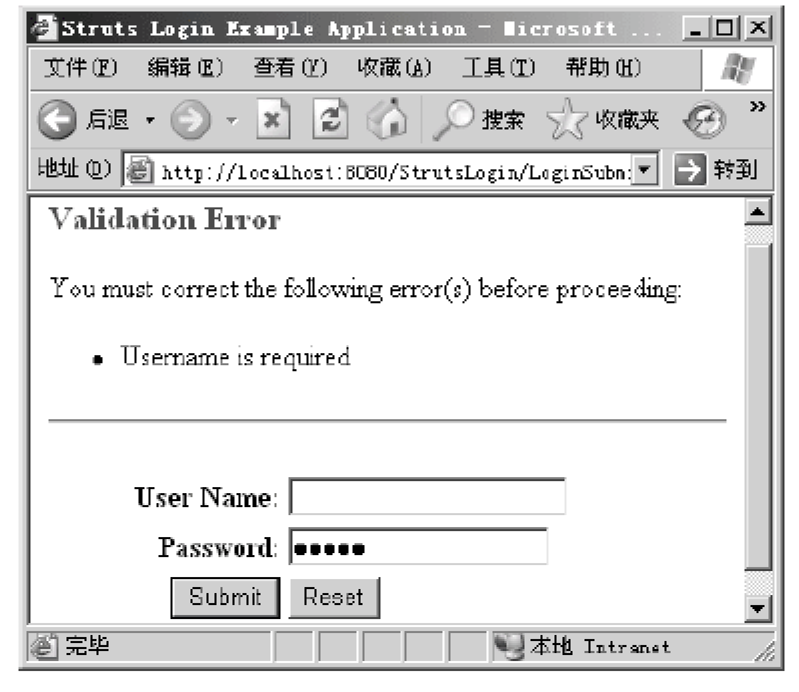


图 19.13　用户名为空时验证失败

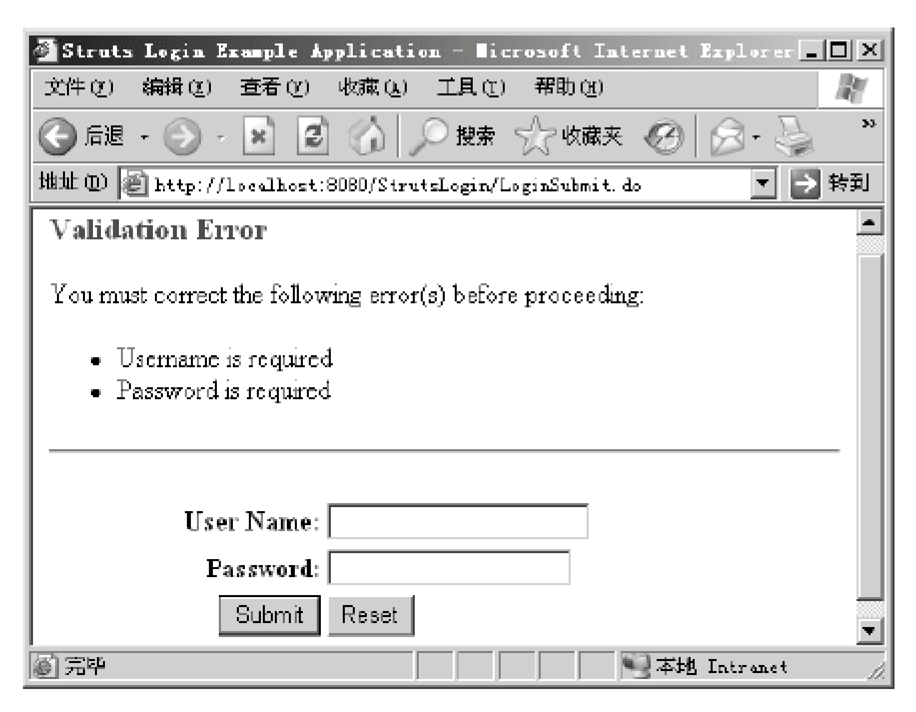


图 19.14　用户名和密码都为空时验证失败

（5）如果用户名和密码都不为空时，表单验证通过，控制器把请求转发给客户请求的资源。

2．用户登录验证

如果用户的表单验证失败，控制器就根据配置文件转发请求并显示错误，如果通过了，控制器把请求转发给客户请求的资源，进行如下的流程：

（1）在上面的例子中，如果客户的表单验证通过了，控制器会把请求转发给 LoginSubmit.do。

（2）Tomcat 根据 web.xml 文件中的定义，把这个请求交给 ActionServlet 处理，ActionServlet 根据 struts-config.xml 文件中的定义，把客户请求交给 cn.ac.ict.LoginAction 类处理。

（3）ActionServlet 创建一个 LoginAction 实例，然后调用这个对象的 execute 方法。

（4）在 execute 方法调用了 isUserLogin 方法验证用户的名称和密码是否被允许登录后，实际的操作过程在 UserInfo 类中完成，具体实现过程可以参考 19.3.4 节中的定义。

（5）如果用户身份得到确认，也就被允许登录了，LoginAction 对象将把用户的信息存到 Session 的一个对象中，并返回成功标志。

（6）ActionServlet 从 action 的属性中查找验证成功时需要转发的页面，看到 action 的如下定义：

```
<action
    path="/LoginSubmit"
    type="cn.ac.ict.LoginAction"
    name="loginForm"
    scope="request"
    validate="true"
    input="/pages/Login.jsp">
```

```
            <forward
                name="success"
                path="/pages/Welcome.jsp"/>
        </action>
```

也就是成功时把请求转发到 Welcome.jsp 页面，此时的 Welcome.jsp 显示如图 19.15 所示。

（7）如果用户不被允许登录（没有合法用户名或密码错误），LoginAction 对象生成一个错误对象，并把它保存到 request 对象中，然后把请求转发，目标是 action 元素的 input 属性指定的页面，也就是 Login.jsp，此时显示页面如图 19.16 所示。



图 19.15　用户验证成功



图 19.16　Login.jsp

3．用户退出

在图 19.15 的页面中单击 Log out 链接，处理的流程与上面差不多，只是把请求交给 cn.ac.ict.LogoutAction 处理，在这个类中，把客户信息从 Session 中清除，并把请求转发给 Welcome.jsp。此时页面显示如图 19.11 所示。

19.5　小　　结

在第 18 章中介绍了一点模型－视图－控制器（MVC）的相关知识，在这一章中介绍了一个经典的 MVC 模式的实现——Struts。

Struts 的控制器由 ActionServlet 和 Action 来担任，Model 模型主要由 ActionForm 和系统状态 Beans 和商业逻辑 Beans 构成，而视图部分大部分为 JSP 页面，也可以是 HTML 页面。

读者学习的重点应该在了解各个组件作用以及如何使用它们，多实践一些例子是很有必要的，读者在搞懂 Struts 后，也可以参考后面几章的介绍，看看几种 MVC 模式实现的共同之处。

第 20 章　MVC 模式实现——WebWork2

在 19 章中介绍了 Struts，在本章将介绍一种 MVC 的实现——WebWork2。WebWork 是由 OpenSymphony 组织开发的，致力于组件化和代码重用的拉出式 MVC 模式 J2EE Web 框架。但现在 WebWork 已经被拆分成了 XWork1 和 WebWork2 两个项目，本书只是介绍 WebWork2。

20.1　快速体验 WebWork2——简单应用实例

20.1.1　WebWork2 简介

WebWork 是由 OpenSymphony 组织开发的，致力于组件化和代码重用的拉出式 MVC 模式 J2EE Web 框架。

WebWork 目前最新版本是 2.1，现在的 WebWork2.x 前身是 Rickard Oberg 开发的 WebWork，但现在 WebWork 已经被拆分成了 XWork1 和 WebWork2 两个项目，如图 20.1 所示。

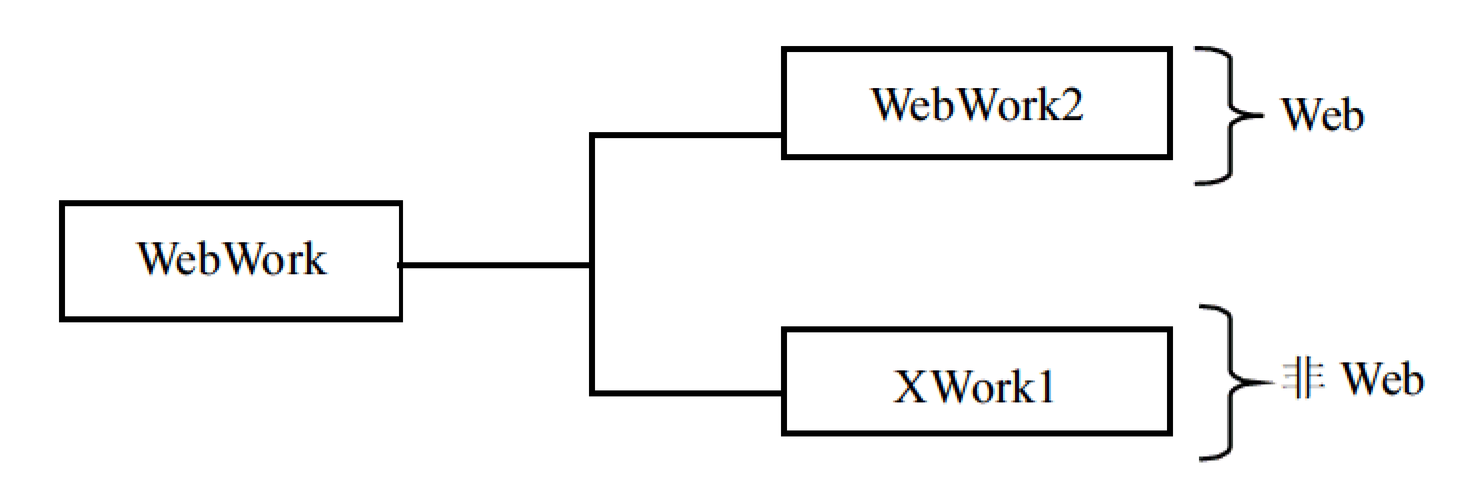


图 20.1　WebWork 被拆分成了 Xwork1 和 WebWork2 两个项目

XWork 简洁、灵活功能强大，它是一个标准的 Command 模式实现，并且完全从 Web 层脱离出来。XWork 提供了很多核心功能：前端拦截机（interceptor）、运行时表单属性验证、类型转换、强大的表达式语言（the Object Graph Notation Language，OGNL）和 IoC（Inversion of Control 倒置控制）容器等。

WebWork2 建立在 XWork 之上，用于处理 HTTP 的响应和请求。WebWork2 使用 ServletDispatcher 将 HTTP 请求变成 Action（业务层 Action 类）、Session（会话）、Application（应用程序）范围的映射和 request 请求参数映射。WebWork2 支持多视图表示，视图部分可以使用 JSP、Velocity、FreeMarker、JasperReports 和 XML 等。在 WebWork2 中添加了对

AJAX 的支持，这些支持构建在 DWR 与 Dojo 这两个框架的基础之上。

20.1.2　建立 WebWork2 开发环境

WebWork2 可以从其官方网站 http://www.opensymphony.com/webwork/免费下载。可按照如下步骤建立 WebWork2 环境。

（1）将下载后的 webwork-2.1.7.zip 文件解压缩到本地硬盘，设定其解压目录为<Webwork2_HOME>。

（2）把<Webwork2_HOME>目录下的 webwork-2.1.7.jar 文件和<Webwork2_HOME>\lib\core 目录下的 JAR 文件复制到 Web 应用的 WEB-INF\lib 目录下。

（3）这样后就可以 Web 应用使用 WebWork2 了。下面演示发布一个 WebWork2 自带的例子。

把<Webwork2_HOME>\webwork-example.war 文件复制到<TOMAT_HOME>\webapps 目录下，启动 Tomcat，在浏览器地址栏中输入地址：http://localhost:8080/webwork-example/，可以看到页面显示如图 20.2 所示。

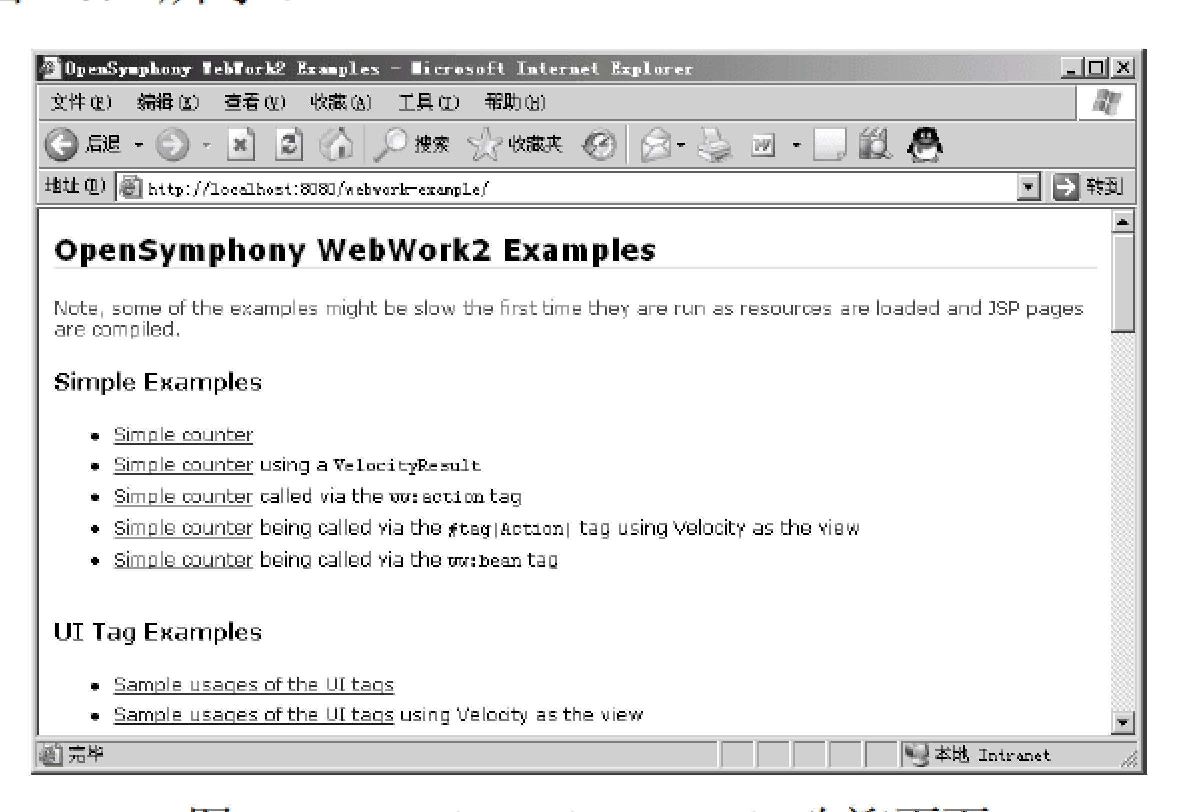


图 20.2　WebWork-Example 欢迎页面

20.1.3　实例介绍

在本节要介绍的例子是一个用户登录的例子，在这个例子中，用户在登录页面中输入用户名和密码，提交后由服务器端的程序验证用户名和密码是否正确，如果正确就转向欢迎界面，否则在登录界面显示错误信息，在这个简单的例子中，贯穿了 WebWork 框架的大部分应用技术，读者可以对 WebWork2 有个大致的认识。

20.1.4　开发构成类和 JSP 文件

1．视图——JSP 页面

首先来看本实例需要使用的登录信息输入页面 index.jsp。在这个页面中用于接受用户的输入，如果用户的用户名和密码正确，将会转发到 main.jsp 视图，否则在登录页面显示错

误信息，其完整代码如下：

```
<%@ page pageEncoding="gb2312" contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>
<html>
<body>
		<form action="login.action">
		<p align="center">登录<br>
		<ww:if test="loginInfo.errorMessage != null">
			<font color="red">
				<ww:property value="loginInfo.errorMessage"/>
			</font>
		</ww:if>
		</p>
		用户名:
			<input type="text" name="loginInfo.username" />
		<br>
		密  码 :
			<input type="password" name="loginInfo.password" />
		<br>
		<p align="center">
			<input type="submit" value="提交" />
			<input type="reset" value="重置" />
		</p>
		</form>
</body>
</html>
```

在这个文件中使用如下语句首先声明使用 WebWork2 的标签库：

```
<%@ taglib prefix="ww" uri="webwork"%>
```

然后在后面的部分就可以使用了，在这个代码中使用了 if 和 property 两个标签：

```
		<ww:if test="loginInfo.errorMessage != null">
		<font color="red">
		<ww:property value="loginInfo.errorMessage"/>
		</font>
		</ww:if>
```

if 标签用来判断是否有错误信息，如果有就使用 property 标签显示出来。

当表单被提交时，浏览器会以两个文本框的值作为参数，向 Web 请求以 login.action 命名的服务。标准 HTTP 协议中并没有.action 结尾的服务资源。需要在 web.xml 中加以设定，这将在配置文件部分介绍。

这个页面显示如图 20.3 所示。



图 20.3　登录信息显示页面

上面提到，如果用户的用户名和密码正确，将会转发到 main.jsp 视图，下面是 main.jsp 文件的完整代码：

```
<%@ taglib prefix="ww" uri="webwork"%>
<html>
<body>
<p align="center">Login Success!</p>
<p align="center">Welcome!
    <ww:property value="#session['username']"/>
</p>
<p align="center">
<!------迭代显示所有的消息------>
  <ww:iterator value="loginInfo.messages" status="index">
    <ww:property/>
  </ww:iterator>
</p>
</body>
</html>
```

这个页面从 Session 中获取用户名并显示处理，随后是使用 WebWork2 的 iterator 标签迭代显示所有的消息：

```
<ww:iterator value="loginInfo.messages" status="index">
<ww:property/>
</ww:iterator>
```

2．配置文件

上面讲到需要在 web.xml 文件中配置如何处理.action 结尾的服务资源，这是通过一个控制器 Servlet 来实现的，web.xml 文件的完整代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
        <display-name>Simple WebWork2 Application</display-name>
        <servlet>
<!------定义 Servlet------>
                <servlet-name>webwork</servlet-name>
<servlet-class>com.opensymphony.webwork.dispatcher.ServletDispatcher</servlet-class>
          <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
<!------对 Servlet 进行映射------>
                <servlet-name>webwork</servlet-name>
                <url-pattern>*.action</url-pattern>
        </servlet-mapping>
<!------声明使用的标签库------>
        <taglib>
                <taglib-uri>webwork</taglib-uri>
       <taglib-location>/WEB-INF/lib/webwork-2.1.7.jar</taglib-location>
        </taglib>
</web-app>
```

从上面的配置文件可以看到，对.action 结尾服务资源进行处理的是一个 Servlet，它就

是 com.opensymphony.webwork.dispatcher.ServletDispatcher，也就是 WebWork2 的控制器，关于这个 Servlet 在后面会详细介绍。

在配置文件中除了配置一个 Servlet 外，还定义声明了一个 Tablib，也就是在 index.jsp 和 main.jsp 页面中使用的标签库。

除了 Web 应用共同的配置文件外，WebWork2 还需要一个配置文件来定义页面导航等信息，也就是 xwork.xml 文件，这个文件必须存放在 Web 应用的 WEB-INF\classes 目录下，本实例的 xwork.xml 文件代码如下：

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-1.0.dtd">

<xwork>
    <include file="webwork-default.xml" />

    <package name="cn.ac.ict" extends="webwork-default">
<!------定义拦截器------>
        <default-interceptor-ref name="defaultStack" />
<!------定义 action 组件------>
        <action name="login" class="cn.ac.ict.LoginAction">
                <result name="success" type="dispatcher">
                    <param name="location">/main.jsp</param>
                </result>
                <result name="loginfail" type="dispatcher">
                    <param name="location">/index.jsp</param>
                </result>
            <interceptor-ref name="params" />
            <interceptor-ref name="model-driven"/>
        </action>

    </package>

</xwork>
```

在这个配置文件中通过 include 元素，可以将其他配置文件导入到默认配置文件 xwork.xml 中，从而实现良好的配置划分。这个文件导入了 WebWork 提供的默认配置 webwork-default.xml（位于 webwork.jar 的根路径）文件。

在 XWork 中，可以通过 package 对 Action 进行分组。类似 Java 中 package 和 class 的关系。为可能出现的同名 Action 提供了命名空间上的隔离。同时，package 还支持继承关系。在这里的定义中可以看到：

```
extends="webwork-default"
```

“webwork-default”是 webwork-default.xml 文件中定义的 package，这里通过继承，“default”package 自动拥有“webwork-default”package 中所定义的关系。

使用 action 配置元素，可以设定 Action 的名称和对应实现类。

通过 result 元素，可以定义 Action 返回语义，即根据返回值，决定处理模式以及响应界面。

3. 控制器 Action

每个请求的动作都对应于一个相应的 Action，一个 Action 是一个独立的工作单元和控制命令，它必须直接或者间接地实现 Xwork 中的 Action 接口，实现 Action 接口的 execute()方法。在本例中使用的一个 Action 的完整代码如下：

```
package cn.ac.ict;
import java.util.Map;
import com.opensymphony.xwork.Action;
import com.opensymphony.xwork.ActionContext;
public class LoginAction implements Action {
	private final static String LOGIN_FAIL="loginfail";

	LoginInfo loginInfo = new LoginInfo();

	public String execute() throws Exception {
		if
(loginInfo.getUsername().equalsIgnoreCase("JSPuser")&loginInfo.getPassword().equals("pass")) {
			//将当前登录的用户名保存到 Session
			ActionContext ctx = ActionContext.getContext();
			Map session = ctx.getSession();
			session.put("username",loginInfo.getUsername());
			//出于演示目的，通过硬编码增加通知消息以供显示
			loginInfo.getMessages().add("Hello JSPuser");
			loginInfo.getMessages().add("Welcome Here!");
			loginInfo.getMessages().add("Thanks!");
			return SUCCESS;
			}else{
				loginInfo.setErrorMessage("Username/Password Error!");
				return LOGIN_FAIL;
				}
		}
	public LoginInfo getLoginInfo() {
		return loginInfo;
		}
	}
```

在这个类的 execute()方法中执行了一些简单的处理，首先获取 Session 并把用户的名字作为一个属性保存在 Session 中，并返回一些欢迎消息。

读者可能注意到在上面的 Action 中使用了一个 LoginInfo 的对象，它封装了所有用户请求和系统响应的信息，就是一个 JavaBeans，代码如下：

```
package cn.ac.ict;

import java.util.ArrayList;
import java.util.List;

public class LoginInfo {
//JavaBeans 的属性
	private String password;
	private String username;
```

```
    private List messages = new ArrayList();
    private String errorMessage;
    //下面是属性的设置和获取方法
    public List getMessages() {
        return messages;
        }
    public String getErrorMessage() {
        return errorMessage;
        }
    public void setErrorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
        }
    public String getPassword() {
        return password;
        }
    public void setPassword(String password) {
        this.password = password;
        }
    public String getUsername() {
        return username;
        }
    public void setUsername(String username) {
        this.username = username;
        }
}
```

20.1.5　编译发布 Web 应用

读者按照上面的介绍准备好所有的文件后，就可以按照如下步骤发布这个 Web 应用：

（1）编译 Action 和 JavaBeans，需要把 WebWork2 的支持 JAR 文件加入到类路径中。

（2）准备好的 Web 应用的结构如图 20.4 所示。

（3）启动 Tomcat，在浏览器地址栏中输入如下地址：http://localhost:8080/20/，随意输入一些信息后提交可以看到页面显示如图 20.5 所示。

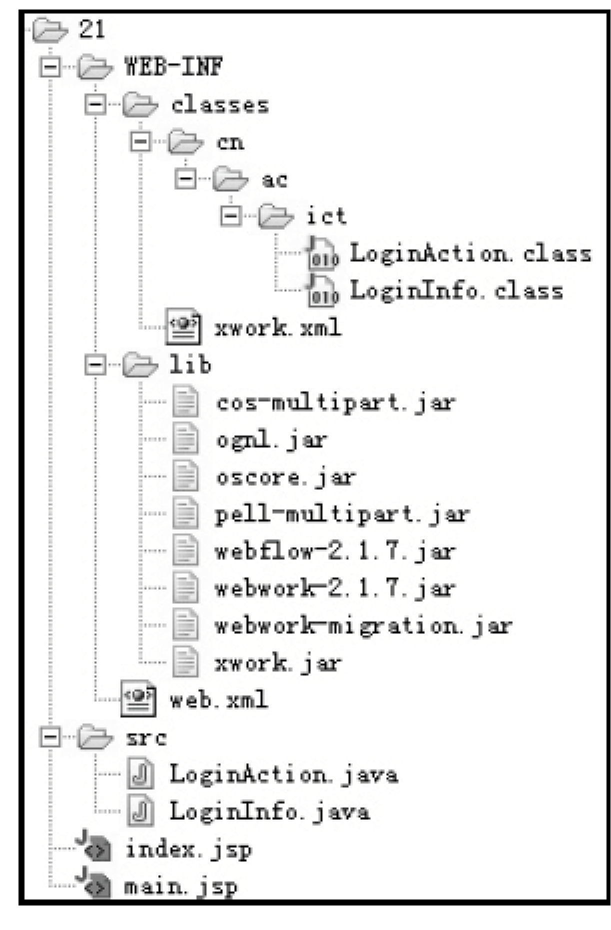


图 20.4　Web 应用目录结构



图 20.5　错误的用户名和密码

输入正确的用户名和密码：JSPuser/pass，提交后，可以看到页面显示如图 20.6 所示。

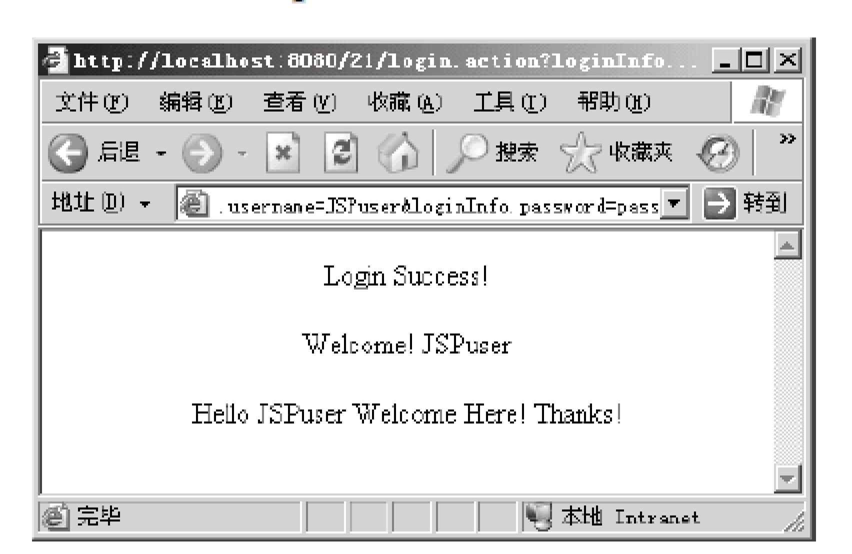


图 20.6　正确的用户登录后的欢迎界面

20.2　WebWork2 组件介绍

WebWork2 首先是一个 MVC 设计模式的实现，所以它有 MVC 结构的共性，同时它也有独特的特点。WebWork2 中比较重要的几个概念有：控制器（ServletDispatcher）、Action 上下文（ActionContext）、动作（Action）和拦截器（Interceptor）。

WebWork2 的基本执行流程如下：

ServletDispatcher 接受客户端的 HTTP 请求，将 HTTP 的相关参数封装后，再传给 XWork。XWork 解析 xwork.xml（配置文件）并根据配置，创建对应的 Action，然后组装并调用相应的拦截器，执行 Action，返回执行结果。

注意：各个组件具体完成的功能见后面几节对各个组件的介绍。

20.2.1　使用 Action 组件响应客户请求

Action 在 MVC 模式中担任控制部分的角色，在 WebWork 中使用得最多。每个请求的动作都对应于一个 Action，一个 Action 是一个独立的工作单元和控制命令，它必须要实现 XWork 中的 Action 接口，实现 Action 接口的 execute()方法。Action 接口的代码如下：

```
package com.opensymphony.xwork;
import java.io.Serializable;
public interface Action extends Serializable {

    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";

    public String execute() throws Exception;
}
```

excute()方法是 Action 类中最重要的部分，它执行后返回 String 类型的值，在 Action 中

返回的值一般使用 Action 接口中定义的标准静态字符常量。例如，前面的 LoginAction 返回的就是 SUCCESS 字符常量，真正的值当然就是“success”，它与 XWork 配置文件中 result 标签 name 的值是相对应的。它用来决定 execute()方法执行完成之后，调用哪一种返回结果。字符常量的含义如下：

- SUCCESS：表示 Action 正确地执行完成，返回相应的视图。
- NONE：表示 Action 正确地执行完成，但并不返回任何视图。
- ERROR：表示 Action 执行失败，返回到错误处理视图。
- INPUT：Action 的执行需要从前端界面获取参数时，INPUT 就是代表这个参数输入的界面，一般在应用中会对这些参数进行验证，如果验证没有通过，将自动返回到该视图。
- LOGIN：Action 因为用户没有登录的原因没有正确执行，将返回该登录视图，要求用户进行登录验证。

Action 根据 FormBean 的不同可以分为两类。

- Field-Driven（字段驱动的）Action。

Action 将直接用自己的字段来充当 FormBean 的功能，上面的例子就是使用这种方式。它一般在页面表单比较简单的情况下使用，而且可以直接用域对象作为 Action 的字段，这样就不用再另写 FormBean，减少了重复代码。

- Model-Driven（模型驱动的）Action。

它很像 Struts 的 FormBean，但在 WebWork 中，只要普通 Java 对象就可以充当模型部分。Model-Driven（模型驱动的）Action 要求 Action 实现 com.opensymphony.xwork.ModelDriven 接口，它有一个方法：Object getModel();，用这个方法返回模型对象就可以了。

20.2.2　使用 ActionContext 获取用户请求信息

ActionContext（com.opensymphony.xwork.ActionContext）是 Action 执行时的上下文，上下文可以看作是一个容器（这里的容器是一个 Map 对象），它存放的是 Action 在执行时需要用到的对象，有请求的参数（Parameter）、会话（Session）、Servlet 上下文（ServletContext）和本地化（Locale）信息等。例如，如果需要取得 request 请求参数“product_id”的值就要按照如下步骤操作：

```
ActionContext context = ActionContext.getContext();
Map params = context.getParameters();
String product_id = (String) params.get("product_id");
```

先获取一个 ActionContext 的对象 Context，然后从 Context 对象中获取所有的请求参数，得到一个 Map 对象（WebWork 框架将与 Web 相关的很多对象重新进行了包装），然后从这个 Map 对象中就可以得到需要的请求参数值。

在每次执行 Action 之前都会创建新的 ActionContext，ActionContext 是线程安全的，也就是说在同一个线程中 ActionContext 里的属性是惟一的，这样 Action 就可以在多线程中使用。通过 ActionContext 的静态方法：ActionContext.getContext()来取得当前的 ActionContext

对象。

如果 Action 需要直接与 JavaServlet 的 HttpSession、HttpServletRequest 等一些对象进行操作，就要使用 ServletActionContext。

ServletActionContext（com.opensymphony.webwork. ServletActionContext），这个类直接继承了 ActionContext，它提供了直接与 JavaServlet 相关对象访问的功能，它可以取得的对象有：

- javax.servlet.http.HttpServletRequest：HTTPservlet 请求对象。
- javax.servlet.http.HttpServletResponse：HTTPservlet 相应对象。
- javax.servlet.ServletContext：Servlet 上下文信息。
- javax.servlet.ServletConfig：Servlet 配置对象。
- javax.servlet.jsp.PageContext：HTTP 页面上下文。

20.2.3　使用 ServletDispatcher 分发客户请求

ServletDispatcher 是默认的处理 Web HTTP 请求的调度器，它是一个 JavaServlet，是 WebWork 框架的控制器。

所有对 Action 调用的请求都将通过这个 ServletDispatcher 调度。这将在 web.xml 中配置 ServletDispatcher 时指定，使所有对 WebWork 的 Action（默认的是.action 的后缀）的请求都对应到该调度的 JavaServlet 中。

ServletDispatcher 接受客户端的 HTTP 请求，将 JavaServlet 的相关对象进行包装，然后传给 XWork 框架，ServletDispatcher 的主要功能调用如下：

1．init()方法

init()方法在服务器启动时调用，它要完成的工作如下：

- 初始化 Velocity 引擎。
- 检查是否支持配置文件重新载入功能。如果 webwork.configuration.xml.reload 设置为 true，每个 request 请求都将重新装载 xwork.xml 配置文件。这在开发环境中特别方便，但在生产环境必须设置为 false。
- 设置一些文件上传的信息，比如：上传的临时目录、上传的最大字节等。都设置在 webwork.properties 文件中，如果在 classpath 中没有找到，它会读取默认的 default.properties。

2．service()方法

每次客户端的请求都将调用 service()方法，它需要完成的工作包括：

- 通过 request 请求取得 Action 的命名空间（namespace，与 xwork.xml 配置文件中 package 标签的 name 对应），例如：

```
/foo/bar/MyAction.action，取得的命名空间为/foo/bar
```

在 xwork.xml 配置文件中应该有这一段：

```
<package name="foo.bar" ……
```

- ❑ 根据 Servlet 请求的 Path，解析出要调用该请求的 Action 的名字（actionName）。
- ❑ 创建 Action 上下文（extraContext）。

前面介绍的 ActionContext 上下文的对象，就是在这里设置的。它将 JavaServlet 相关的对象进行包装，放入到这个 Map 对象中，并返回这个 Map 对象。

- ❑ 根据前面获得的 namespace、actionName、extraContext（一个 Map 对象），创建一个 ActionProxy，代码如下：

```
ActionProxy proxy = ActionProxyFactory.getFactory().createActionProxy(namespace,actionName,
extraContext);
```

默认的 proxy 是 com.opensymphony.xwork.DefaultActionProxy。

- ❑ 执行 proxy 的 execute()方法。

这个方法最核心的语句是：retCode = invocation.invoke();，invocation 对象的 invoke()方法遍历并执行这个 Action 对应的所有拦截器，然后执行 Action 对应的方法（默认的是 execute()），根据 Action 执行的返回值去调用相应的 Result（返回结果处理）的方法。

20.2.4　使用 Interceptor（拦截器）框架

Interceptor（拦截器）将 Action 共用的行为独立出来，在 Action 执行前后运行。它的执行类似于 Servlet 过滤器。

Interceptor 将很多功能从 Action 中独立出来，大量减少了 Action 的代码，独立出来的行为具有很好的重用性。XWork、WebWork 的许多功能都是由 Interceptor 实现的，可以在配置文件中组装 Action 用到的 Interceptor，它会按照指定的顺序，在 Action 执行前后运行。Interceptor 在框架中的应用如图 20.7 所示。

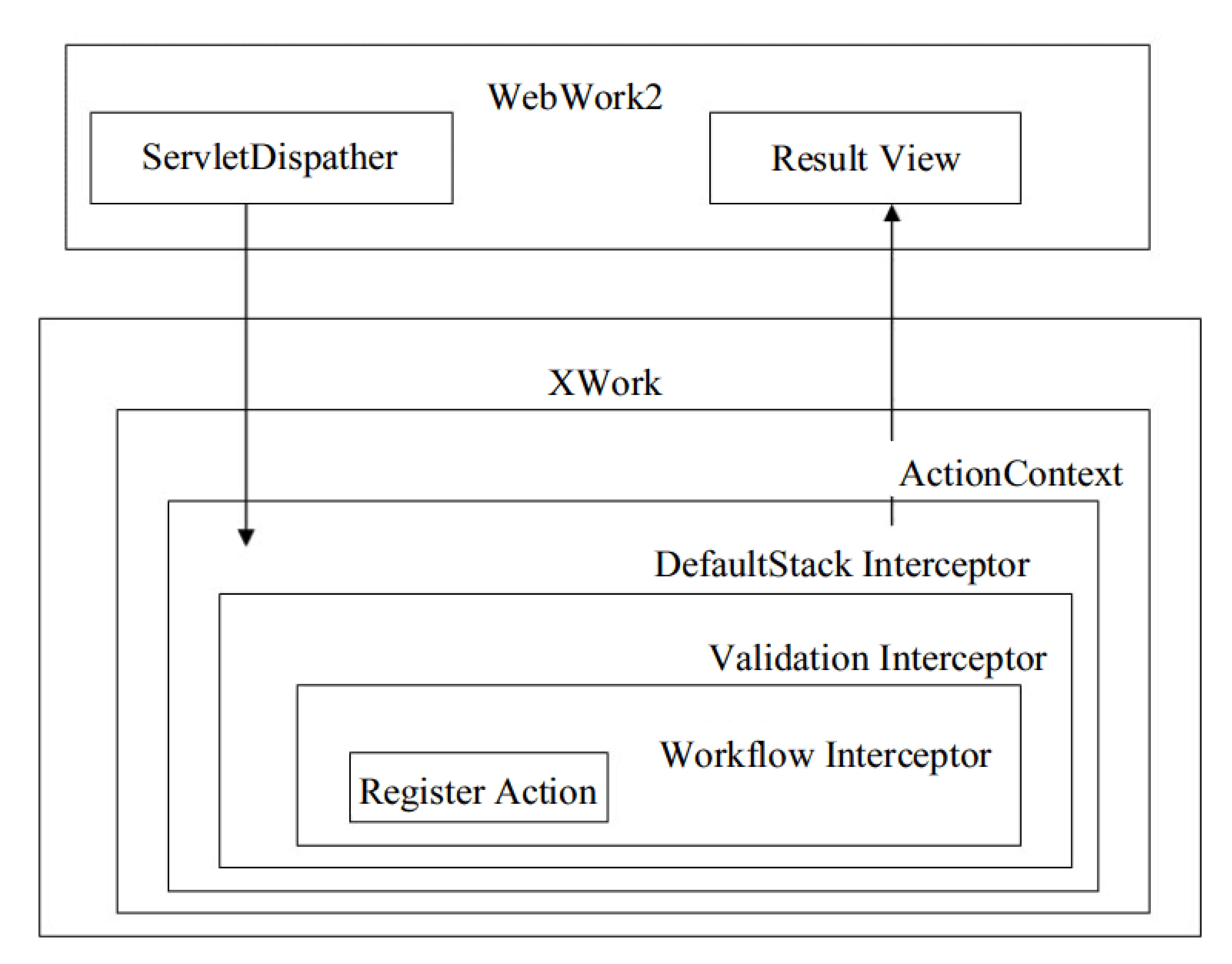


图 20.7　WebWork2 框架

当客户提交对 Aciton 的请求时，ServletDispatcher 会根据客户的请求去调度并执行相应的 Action。在 Action 执行之前，调用被 Interceptor 截取，也就是 Interceptor 在 Action 执行前后运行。

所有 XWork 提供的 Interceptor 都在 webwork-default 中有定义，使用 Interceptor 可以完成计时、输出日志和对用户进行验证等功能，框架中给开发者供了很多实用的 Interceptor，它们的具体功能如下：

- timer：记录 Action 执行的时间，并作为日志信息输出。
- logger：在日志信息中输出要执行的 Action 信息。
- chain：将前一个执行结束的 Action 属性设置到当前的 Action 中。它被用在 ResultType 为“chain”指定结果的 Action 中，该结果 Action 对象会从 OgnlValueStack 中获得前一个 Action 对应的属性，它实现 Action 链之间的数据传递。
- static-params：将 xwork.xml 配置文件中定义的 Action 参数，设置到对应的 Action 中。Action 参数使用<param />标签，是<action />标签的直接子元素。这里定义的 Action 类必须实现 com.opensymphony.xwork.config.entities. Parameterizable 接口。
- params：将 request 请求的参数设置到相应 Action 对象的属性中。
- model-driven：如果 Action 实现 ModelDriven 接口，它将 getModel()取得的模型对象存入 OgnlValueStack 中。
- component：激活组件功能支持，让注册过的组件在当前 Action 中可用，即为 Action 提供 IoC（依赖倒转控制）框架的支持。
- token：核对当前 Action 请求（request）的有效标识，防止重复提交 Action 请求（request）。
- token-session：功能同上，但当提交无效的 Action 请求标识时，它会将请求数据保存到 Session 中。
- validation：实现使用 XML 配置文件（{Action}-validation.xml）对 Action 属性值进行验证，详细请看后面介绍的验证框架。
- workflow：调用 Action 类的验证功能，假设 Action 使用 ValidationAware 实现验证（ActionSupport 提供此功能），如果验证没有通过，workflow 会将请求返回到 input 视图（Action 的<result />中定义的）。
- servlet-config：提供 Action 直接对 HttpServletRequest 或 HttpServletResponse 等 JavaServlet api 的访问，Action 要实现相应的接口，例如：ServletRequestAware 或 ServletResponseAware 等。如果必须要提供对 JavaServlet API 的访问，建议使用 ServletActionContext，在前面 ActionContext 章节中有介绍。
- prepare：在 Action 执行之前调用 Action 的 prepare()方法，这个方法是用来准备 Action 执行之前要做的工作。它要求 Action 必须实现 com.opensymphony.xwork. Preparable 接口。
- conversionError：用来处理框架进行类型转化（Type Conversion）时的出错信息。它将存储在 ActionContext 中的类型转化（Type Conversion）错误信息转化成相应的 Action 字段的错误信息保存在堆栈中。根据需要可以将这些错误信息在视图中

显示出来。

除了使用系统提供的拦截器，也可以自定义拦截器，使用自定义拦截器的步骤如下：

- ❑ 创建一个自己需要的 Interceptor 类，它必须实现 com.opensymphony.xwork.interceptor.Interceptor 接口。
- ❑ 在配置文件（xwork..xml）中申明这个 Interceptor 类，它放在标签<interceptor />中，并把 interceptor />标签嵌入在<interceptors />标签内部。
- ❑ 创建 Interceptor 栈，使用标签<interceptor-stack />，让一组 Interceptor 可以按次序调用（可选）。

指定 Action 所要用到的 Interceptor，可以用<interceptor-ref />或<default-interceptor-ref />标签。<interceptor-ref />标签指定某个 Action 所用到的 Interceptor，如果 Action 没有被用<interceptor-ref />指定 Interceptor，它将使用<default-interceptor-ref />指定的 Interceptor。

20.2.5　使用验证框架验证用户输入

WebWork2 提供了在 Action 执行之前对输入数据验证的功能，它使用了其核心 XWork 的验证框架。主要提供了如下功能：

- ❑ 可配置的验证文件。它的验证文件是一个独立的 XML 配置文件，对验证的添加、修改时只需要修改配置文件，无须编译任何的 Class。
- ❑ 验证文件和被验证的对象无关联。验证对象是普通的 JavaBeans 即可（可以是 FormBean、域对象等），不需实现任何额外的方法或继承额外的类。
- ❑ 多种不同的验证方式。因为验证功能是可以继承的，所以可以用多种不同的方式指定验证文件，比如：通过父类的 Action、通过 Action、通过 Action 的方法、通过 Action 所使用的对象等。
- ❑ 强大的表达式验证。它使用了 OGNL（Object-Graph Navigation Language，是一种功能强大的表达式语言）的表达式语言，提供强大的表达式验证功能。
- ❑ 同时支持服务器端和客户端验证。

WebWork 为不同的验证要求提供不同的验证类型。一个验证类型，一般是有一个类来提供。这个类必须实现接口：com.opensymphony.xwork.validator.Validator，但我们在写自己的验证类型时，无须直接实现 Validator 接口，它有抽象类可供直接使用，如 ValidatorSupport、FieldValidatorSupport 等。

验证类型在使用之前，必须要在 ValidatorFactory（com.opensymphony.xwork.validator.ValidatorFactory）中注册。可以有两种方法实现验证类型的注册。

- ❑ 写程序代码进行注册，它使用 ValidatorFactory 类的静态方法：registerValidator(String name, String className)。
- ❑ 使用配置文件 validators.xml 进行注册，要求把文件 validators.xml 放到 ClassPath 的根目录中（/WEB-INF/classes）。

在实际开发中，一般都使用第二种注册方法。

如果 Action 要使用验证框架的验证功能，它必须在配置文件中指定拦截器“validation”，

它的定义如下：

```
<interceptor name="validation" class="com.opensymphony.xwork.validator.ValidationInterceptor"/>.
```

验证文件必须以 ActionName-validation.xml 格式命名，它必须被放置到与这个 Action 相同的包中。也可以为这个 Action 通过别名的方式指定验证文件，它的命名格式为：ActionName-aliasname-validation.xml。其中“ActionName”是 Action 的类名；“aliasname”是在配置文件（xwork.xml）中定义这个 Action 所用到的名称。这样，同一个 Action 类，在配置文件中的不同定义就可以对应不同的验证文件。验证框架也会根据 Action 的继承结构去查找 Action 的父类验证文件，如果找到它会去执行这个父类的验证。

在 20.3 节中介绍的例子就使用了这样的一个验证。

20.2.6　配置 XWork

XWork 配置文件是以 XWork 命名的.xml 文件，它必须放到类路径（classPath）的根目录——WEB-INF\classes 目录中。

这个文件定义了 Action、Interceptor、Result 的配置和相互之间的映射。下面是一个 XWork 配置文件的例子：

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-1.0.dtd">

<xwork>
        <!-- Include webwork defaults (from WebWork-2.1 JAR). -->
        <include file="webwork-default.xml" />

        <!--  配置默认的包  -->
        <package name="cn.ac.ict" extends="webwork-default">
                <!--定义默认的拦截器栈  -->
                <default-interceptor-ref name="defaultStack" />
                <!--定义 action 组件  -->
            <action name="registerSupport" class="cn.ac.ict.RegisterAction" >
                <result name="success" type="dispatcher">
                     <param name="location">/register-result.jsp</param>
                </result>
                <result name="input" type="dispatcher">
                     <param name="location">/registerSupport.jsp</param>
                </result>
                <interceptor-ref name="validationWorkflowStack"/>
            </action>

        </package>

</xwork>
```

在 xwork.xml 文件中可以配置如下元素：

（1）XWork：XWork 元素配置文件的所有内容，它是配置文件的根元素，它的直接子标签有<package>和<include>，例如在上面例子中的 XWork 元素就包含一个<package>元素和一个<include>元素。

（2）Package：Action、Interceptor、Result 是在此标签中定义，它需要设置的属性如表 20.1 所示。

表 20.1　Package元素属性

属　性	是 否 必 需	描　述
name	是	用来标识package的名称
extends	否	继承它所扩展的package配置信息
namespace	否	指定package的命名空间，默认是" "
abstract	否	声明package是抽象的

（3）Result-type：用来定义输出结果类型的 Class，使用名－值对的形式来定义。自己写的输出结果类型也必须在这里定义。

（4）Interceptors：它是一个简单的<interceptors>标签，程序中需要的 Interceptor 和 Interceptor-stack 都在此标签内定义。

- Interceptor：用来定义拦截器。它的定义非常简单，使用名－值对的形式。
- Interceptor-stack：用来将上面定义的 Interceptor 组织成堆栈的形式，这样就可以创建一组标准的 Interceptor，让它按照顺序执行。在 Action 中直接引用这个 Interceptor 堆栈就可以了，不用逐个 Interceptor 去引用。

（5）Global-results：它允许定义全局的输出结果（global result），比如登录页面、操作错误处理页面。只要继承它所在的 package，这些输出结果都是可见的。

（6）Action：用来配置 Action 的名称（name）和它对应的 Class。通过这个 Action 的名称和它所在 package 的 namespace 去配置文件中取得这个 Action 的配置信息。它可以通过<param>来设置参数，Action 在执行时会取得配置文件中设置的参数（通过拦截器 StaticParametersInterceptor）。

Action 可以配置一个或多个输出结果（result）。一个输出结果的名称对应于 Action 执行完成返回的字符串。<result>标签的 type 属性对应前面定义过的 result-type，说明 reslut 的类型，例如在上面的例子中定义了名为 registerSupport 的 Action：

```
<action name="registerSupport" class="cn.ac.ict.RegisterAction" >
    <result name="success" type="dispatcher">
        <param name="location">/register-result.jsp</param>
    </result>
    <result name="input" type="dispatcher">
        <param name="location">/registerSupport.jsp</param>
    </result>
    <interceptor-ref name="validationWorkflowStack"/>
</action>
```

可见，在这个 Action 中就定义了两个 result 元素（一个名为 success，一个名为 input）

和一个默认的拦截器。

（7）Include：xwork.xml 文件可以被分成好几个不同的文件，xwork.xml 通过<include>标签引用被包含的文件，例如：<include file="webwork-default.xml"/>。被包含的文件必须是 package 标签中的内容。如果要继承被包含文件的 package，必须将<include>标签放在其上面，因为配置文件是按照由上而下的顺序解析的。

20.3　使用 WebWork2 开发产品录入系统——WebWork2 实例

在本节介绍一个简单的产品录入系统，重点演示 Webwork2 验证框架的使用。

20.3.1　创建 Action 组件

下面介绍本实例使用的 Action 组件，它通过继承 ActionSupport 类简介实现了 Action 接口，其代码如下：

```
package cn.ac.ict;
import com.opensymphony.xwork.ActionSupport;
public class ProductInputAction extends ActionSupport{
	private Product product= new Product();
	public Product getProduct(){
		return this.product;
	}
	//不作细致处理，返回 SUCCESS
	public String execute(){
		return SUCCESS;

	}
}
```

在这个类中使用了一个 Product 对象，它的代码如下：

```
package cn.ac.ict;
import java.io.Serializable;
import java.util.Date ;
public class Product implements Serializable{
//下面是 JavaBeans 的属性
	private String pname;
	private String pcomp;
	private Date pmadeyear;
	private float price;
	private int amount;

	public Product(){
```

```
        }
//下面是 JavaBeans 属性的设置和获取方法
        public String getPname(){
                return pname;
        }
        public String getPcomp(){
                return pcomp;
        }

        public Date getPmadeyear(){
                return pmadeyear;
        }
        public float getPrice(){
                return price;
        }

        public int getAmount(){
                return amount;
        }

        public void setPname(String productname){
                pname = productname;
        }
        public void setPcomp(String productcomp){
                pcomp = productcomp;
        }

        public void setPmadeyear(Date madeyear){
                pmadeyear = madeyear;
        }
        public void setPrice(float price){
                this.price = price;
        }

        public void setAmount(int pamount){
                amount = pamount;
        }

}
```

这里使用的类都很简单就不介绍了。

20.3.2　创建视图组件

在本实例中使用了两个视图：productinput.jsp（产品信息输入界面）和 productinput_result.jsp（产品信息输入结果页面），下面分别介绍：

1．编写 productinput.jsp 文件——产品信息输入页面

```
<%@ taglib uri="webwork" prefix="ww" %>
```

```
<html>
<head><title>Product Input Example Using WebWork2</title></head>
<body>
<table border=0 width=97%>
<tr><td align="left">
    <ww:form name=" 'test' " action=" 'ProductInputSupport.action' " method=" 'POST' ">
    <ww:textfield label=" 'Product Name' " name=" 'product.pname' " required="true"/>
            <ww:textfield label=" 'Product Made Where' " name=" 'product.pcomp' " required
="true"/>
            <ww:textfield label=" 'Made Date' " name=" 'product.pmadeyear' " required="true"/>
            <ww:textfield label=" 'Price' " name=" 'product.price' "/>
            <ww:textfield label=" 'Amount' " name=" 'product.amount' " required="true"/>
            <ww:submit value=" 'Submit' "/>
        </ww:form>
</td></tr>
</table>
</body>
</html>
```

这是产品录入界面，提供一些输入框来接受用户的输入，这个页面显示如图 20.8 所示。



图 20.8　产品录入界面

2．编写 productinput_result.jsp 文件——产品信息输入结果显示页面

这个页面显示产品输入的结果：

```
<%@ taglib prefix="ww" uri="webwork" %>
<html>
<head><title>Register result</title></head>
<body>
    <table border=0 width=97%>
        <tr>
            <td align="left">
            Congratulation,Input Product success!<p>
            Product name:<ww:property value="product.pname"/><br>
            Product Made Where:<ww:property value="product.pcomp"/><br>
            Made Date:<ww:property value="product.pmadeyear"/><br>
            Price:<ww:property value="product.price"/><br>
            Amount:<ww:property value="product.amount"/><br>
            </td>
        </tr>
```

```
        </table>
</body>
</html>
```

20.3.3　验证框架

根据前面的介绍，在本实例中使用验证框架需要按照如下步骤进行：

（1）使用第二种方法进行验证注册——使用配置文件 validators.xml 进行注册，要求把文件 validators.xml 放到 classpath 的根目录（/WEB-INF/classes）中。

（2）本实例中使用的 validators.xml 文件的内容如下：

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd">

<validators>
        <validator name="required"
class="com.opensymphony.xwork.validator.validators.RequiredFieldValidator"/>

        <validator name="requiredstring"
class="com.opensymphony.xwork.validator.validators.RequiredStringValidator"/>
        <validator name="int"
class="com.opensymphony.xwork.validator.validators.IntRangeFieldValidator"/>

        <validator name="date"
class="com.opensymphony.xwork.validator.validators.DateRangeFieldValidator"/>
        <validator name="expression"
class="com.opensymphony.xwork.validator.validators.ExpressionValidator"/>

        <validator name="fieldexpression"
class="com.opensymphony.xwork.validator.validators.FieldExpressionValidator"/>

        <validator name="email"
class="com.opensymphony.xwork.validator.validators.EmailValidator"/>

        <validator name="url"
class="com.opensymphony.xwork.validator.validators.URLValidator"/>

        <validator name="visitor"
class="com.opensymphony.xwork.validator.validators.VisitorFieldValidator"/>

        <validator name="conversion"
class="com.opensymphony.xwork.validator.validators.ConversionErrorFieldValidator"/>

</validators>
```

（3）编写本实例使用的验证文件，它的名字是 ProductInputAction-validation.xml，前一部分的名字和 Action 的类名是一致的。

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator 1.0//EN" "http://www.
opensymphony.com/xwork/xwork-validator-1.0.dtd">
<validators>
<!----------产品名称为空时提示的信息------------>
    <field name="product.pname">
     <field-validator type="requiredstring">
            <message>You must enter a value for 'Product Name'.</message>
        </field-validator>
    </field>
<!----------产品的生产企业为空时输出的提示信息------------>
    <field name="product.pcomp">
     <field-validator type="requiredstring">
            <message>You must enter a value for 'Product Made Where'.</message>
        </field-validator>
    </field>
    <field name="product.pmadeyear">
<!----------日期输入错误时的提示信息------------>
     <field-validator type="date">
        <message>You must enter a right Date value for 'Made Date'.</message>
        </field-validator>
    </field>
<!----------产品数量输入的范围出现错误时提示的信息------------>
    <field name="product.amount">
     <field-validator type="int">
            <param name="min">6</param>
            <param name="max">100</param>
            <message>Amount must be between ${min} and ${max}, current value is
${product.amount}.</message>
        </field-validator>
    </field>

</validators>
```

（4）如果用户输入的数据验证没有通过，需重新返回输入页面，并给出错误信息提示。拦截器栈“validationWorkflowStack”实现了这个功能。它首先验证用户输入的数据，如果验证没有通过将不执行 Action 的 execute()方法，而是将请求重新返回到输入页面。

20.3.4　编写配置文件

本实例使用的 web.xml 文件与本章第一节的例子中的 web.xml 文件是一样的，这里就不介绍了，下面是 WebWork2 配置文件的 xwork.xml：

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-1.0.dtd">

<xwork>
        <include file="webwork-default.xml" />
<!-- 配置默认的包 -->
<package name="cn.ac.ict" extends="webwork-default">
<default-interceptor-ref name="defaultStack" />
<action name="ProductInputSupport" class="cn.ac.ict.ProductInputAction" >
     <result name="success" type="dispatcher">
          <param name="location">/productinput_result.jsp</param>
     </result>
     <result name="input" type="dispatcher">
          <param name="location">/productinput.jsp</param>
     </result>
     <interceptor-ref name="validationWorkflowStack"/>
</action>

        </package>

</xwork>
```

上面的配置文件中使用了验证拦截器栈来显示错误信息：

```
<interceptor-ref name="validationWorkflowStack"/>
```

20.3.5　运行基于 WebWork2 的 Web 应用

读者按照上面的介绍准备好所有的文件后，就可以按照如下步骤发布这个 Web 应用：

（1）编译 Action 和 JavaBeans，需要把 WebWork2 的支持 JAR 文件加入到类路径中。

（2）准备好的 Web 应用的结构如图 20.9 所示。

（3）启动 Tomcat，在浏览器地址栏中输入如下地址：http://localhost:8080/webwork2/productinput.jsp，输入后提交可以看到页面如图 20.10 所示。

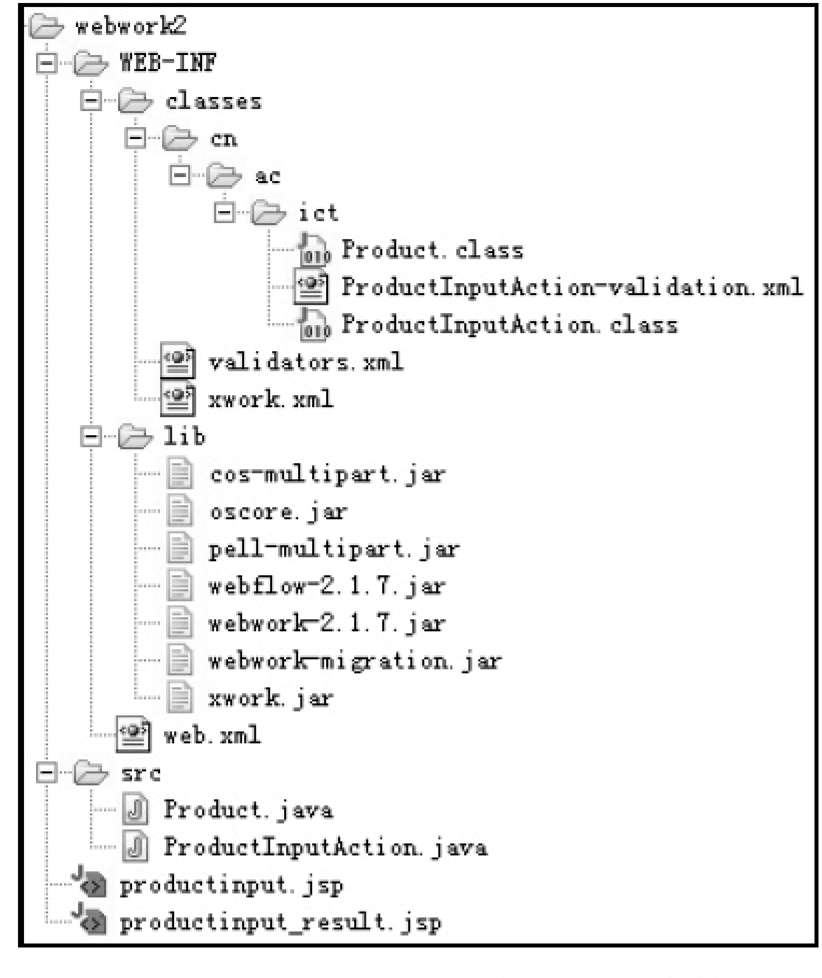


图 20.9　Web 应用的目录结构

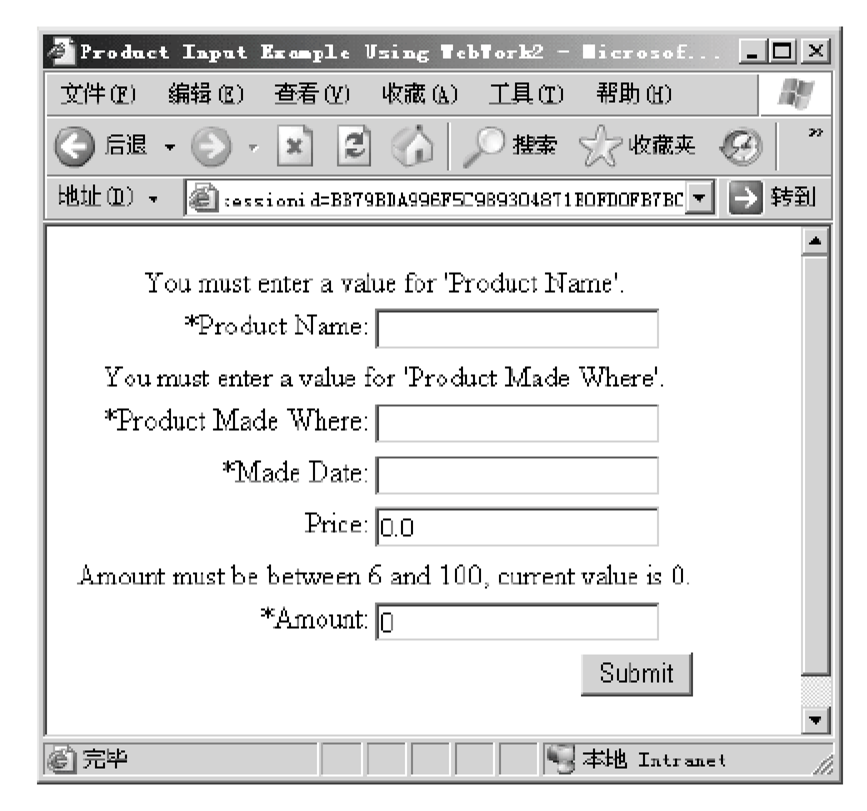


图 20.10　验证框架的实例

可以看到如果用户不输入数据或者输入错误的数据都会提示错误，这些错误信息就是在 ProductInputAction-validation.xml 文件中定义的。

（4）正确地输入数据并提交后，可以看到页面显示如图 20.11 所示。

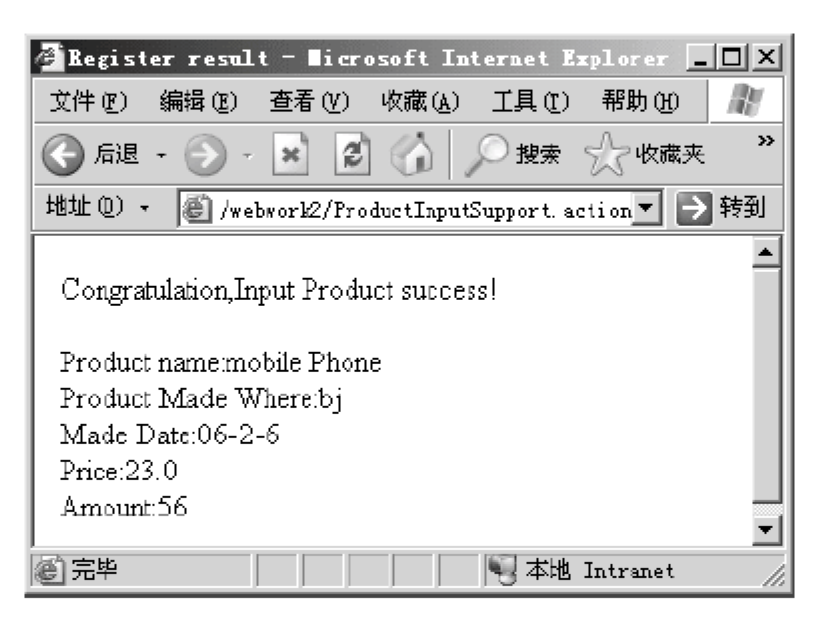


图 20.11　通过验证框架

20.4　小　　结

在第 19 章介绍了 Struts 框架后，本章又介绍了 WebWork2。WebWork2 建立在 XWork 基础之上，处理 HTTP 的响应和请求，使得它与 Servlet 的 request 和 response 很好地解耦。

WebWork2 中的一些概念如验证器、拦截器等希望读者能重点掌握，难点在于了解 WebWork2 的工作原理，读者在阅读本章基本了解该技术的基础上应该多加实践。

第 21 章　Java Server Faces

Java Server Faces（JSF）是一种 MVC 应用程序框架，用于创建基于 Web 的用户界面。如果读者熟悉 Struts（一种流行的开放源代码的基于 JSP 的 Web 应用程序框架，在本书的“MVC 模式实现——Struts”一章中有介绍）和 Swing（针对桌面应用程序的标准 Java 用户界面），就可以将 Java Server Faces 想象成这两种框架的组合。与 Struts 一样，JSF 通过一个控制器 Servlet 来提供 Web 应用程序生命周期管理；也与 Swing 一样，JSF 提供了一个带有事件处理和组件呈现（rendering）的丰富组件模型。

21.1　快速体验 Java Server Faces——使用 Java Server Faces 开发的简单例子

21.1.1　Java Server Faces 技术介绍

Java Server Faces（JSF）是一种用于构建 Web 应用程序的新标准 Java 框架。它提供了一种以组件为中心来开发 Java Web 用户界面的方法，JSF 开发可以简单到只需将用户界面（UI）组件拖放到页面上，丰富而强健的 JSF API 为“系统开发人员”提供了无与伦比的功能和编程灵活性。

JSF 通过将构建良好的模型—视图—控制器（MVC）设计模式集成到它的体系结构中，确保了应用程序具有更高的可维护性。最后，由于 JSF 是通过 Java Community Process（JCP）开发的一种 Java 标准，因此开发工具供应商完全能够为 Java Server Faces 提供易于使用的、高效的可视化开发环境。

JSF 减轻了开发基于 Web 的应用程序的工作量，因为它具有如下的特性：

- 可以通过一组标准的、可重用的服务器端组件来创建用户界面。
- 提供了一组 JSP 标签以访问这些组件。
- 在表单重新显示时，透明地保存状态信息并重新填充表单。
- 提供了实现自定义组件的框架。
- 封装了事件处理和组件呈现，以便用户可以使用标准的 JSF 组件或自定义组件来支持除 HTML 之外的标记语言。
- 工具开发商可以开发针对标准 Web 应用程序框架的 IDE。

21.1.2　建立 Java Server Faces 开发环境

要使用 Java Server Faces 进行开发，需要到 Sun 的官方网站下载其相关的 JAR 文件，下载地址是 http://java.sun.com/j2ee/javaserverfaces/download.html。下载完成后，将下载的压缩包 jsf-1_1_01.zip 解压到一个临时目录，可以看到在 lib 目录下有多个 JAR 文件，jsf-api.jar 和 jsf-impl.jar 是 Java Server Faces 运行的核心类库，其他的几个 JAR 文件是它用到的组件，同样是不可缺少的，在编译 JSF 相关类时要把这些文件放到类路径中，在发布 Web 应用时应确保 Web 应用可以找到这些 JAR 文件。

21.1.3　编写相关类和文件

1．编写管理 Bean

具体管理 Bean 的作用会在下面介绍，简单而言就是它封装了用户提交的信息并带有对这些信息进行验证的方法，在本例中使用的管理 Bean 的源代码如下：

```
package cn.ac.ict;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.LongRangeValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
import javax.faces.application.FacesMessage;
public class UserRegistrationBean{
//下面是 JavaBeans 的属性
            private String firstName;
            private String lastName;
            private short age;
            private String sex;
            private String phone;
            private String email;
//下面是与 JavaBeans 属性对应的设置和获取方法
            public String getFirstName(){
                  return this.firstName;
            }

            public String getLastName(){
                  return this.lastName;
            }

            public short getAge(){
                  return age;
            }
```

```
		public String getSex(){
			return sex;
		}

		public String getPhone(){
			return phone;
		}

		public String getEmail(){
			return email;
		}
		public void setFirstName(String first){
			firstName = first;
		}

		public void setLastName(String last){
			lastName = last;
		}

		public void setAge(short age){
			this.age = age;
		}

		public void setSex(String sex){
			this.sex = sex;
		}

		public void setPhone(String phone){
			this.phone = phone;
		}

		public void setEmail(String email){
			this.email = email;
		}
		//验证 firstName 的输入是否合法
		public void validateFistName(FacesContext context, UIComponent toValidate,Object
value){

		}
		//验证 age 的输入是否合法，这里什么都不做
		public void validateAge(FacesContext context, UIComponent toValidate,Object value){

		}
		//验证 Email 的输入是否合法
		public void validateEmail(FacesContext context, UIComponent toValidate,Object value)
{
		String email = (String) value;
		if (email.indexOf('@') == -1) {
			((UIInput)toValidate).setValid(false);
```

```
                FacesMessage message = new FacesMessage("Invalid Email");
                context.addMessage(toValidate.getClientId(context), message);
            }
        }
            public String validateRegister(){
                return "success";
            }
}
```

可以看到，它和普通的 JavaBeans 很相似，只是多了几个验证数据的方法，这也就是它与普通 JavaBeans 的区别了。

2．编写 JSP 页面

JSF 是 MVC 模式的一个实现，它可以使用 JSP 页面作为视图，也可以不使用 JSP 页面作为视图，使用起来还是很方便的，在本例中使用 JSP 页面作为视图，代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
  <head>
    <title>User Registration.jsp Form</title>
  </head>

   <body>
    <f:view>
      <h2>User Register Validation </h2>
        <h:form id="userForm">

        <h:messages id="userMessages" showDetail="true" layout="table"/>

          <!-------创建一个 3 列的表格  -->
               <h:panelGrid columns="3">
<!--------可以认为对应于 JavaBeans 的 firstName  属性--------------->
                <h:outputLabel for="firstName" accesskey="F" >
                            <h:outputText value="First Name"/>
                  </h:outputLabel>

                   <h:inputText id="firstName"
                                  value="#{UserRegistrationBean.firstName}"
                                  required="true">
                            <f:validateLength minimum="2" maximum="25" />
                   </h:inputText>

                <h:message
                     style="color: red; text-decoration: overline"
                            id="firstNameError"
                            for="firstName"/>
```

```
<!--------可以认为对应于 JavaBeans 的 lastName 属性---------------->
                <h:outputLabel for="lastName" accesskey="L"
                               id="lastNameLabel"
                               >
                       <h:outputText value="Last Name"/>
                </h:outputLabel>

                <h:inputText id="lastName"
                               value="#{UserRegistrationBean.lastName}"
                               required="true">
                      <f:validateLength minimum="2" maximum="25" />
                </h:inputText>

         <h:message style="color: red; text-decoration: overline" id="lastNameError" for=
"lastName"/>
<!--------可以认为对应于 JavaBeans 的 age 属性---------------->
         <h:outputLabel for="age" accesskey="a" id="ageLabel">
              <h:outputText value="Age" id="text_1"/>
              </h:outputLabel>

         <h:inputText id="age" value="#{UserRegistrationBean.age}">
                     <f:converter converterId="javax.faces.Short"/>

                     <f:validateLongRange maximum="150" minimum="0"/>
                </h:inputText>

         <h:message style="color: red; text-decoration: overline" id="ageError" for="age"/>
<!--------可以认为对应于 JavaBeans 的 sex 属性---------------->
         <h:outputLabel for="sex" accesskey="S" id="sexLabel">
              <h:outputText value="Sex"/>
         </h:outputLabel>

         <h:inputText id="sex" value="#{UserRegistrationBean.sex}"required="true">
              <f:validateLength minimum="2" maximum="10" />
         </h:inputText>

         <h:message style="color: red; text-decoration: overline" id="sexError" for="sex"/>
<!--------可以认为对应于 JavaBeans 的 email 属性---------------->
         <h:outputLabel for="email" accesskey="e"      id="emailLabel">
              <h:outputText value="email"/>
         </h:outputLabel>

         <h:inputText id="email" value="#{UserRegistrationBean.email}"
validator="#{UserRegistrationBean.validateEmail}" >
         </h:inputText>

         <h:message style="color: red; text-decoration: overline" id="emailError" for="email"/>
<!--------可以认为对应于 JavaBeans 的 phone 属性---------------->
         <h:outputLabel for="phone" accesskey="p" id="phoneLabel">
              <h:outputText value="Phone"/>
         </h:outputLabel>
```

```
            <h:inputText id="phone" value="#{UserRegistrationBean.phone}" >
            </h:inputText>

            <h:message style="color: red; text-decoration: overline" id="phoneError" for="phone"/>
            </h:panelGrid> <br/>
                <h:panelGroup>
                    <h:commandButton
                    id="register"
                    action="#{UserRegistrationBean.validateRegister}"
                    value="Register"
                    />
                </h:panelGroup>

      </h:form>
    </f:view>
  </body>
</html>
```

可以看到，在文件一开始，就声明使用两个自定义的标签库，分别是 JSF 的 HTML 标签库和 core 标签库，各个标签与自定义标签是同样的技术，不过标签的使用方法不同而已，读者如果要掌握这些标签，可以查看其相应的 TLD 文件，了解各个标签的使用方法，可以看到这个文件中使用了很多标签，和 HTML 的标签有些相像，但又有不同，这里就不详细介绍了。

另外，在 firstName、lastName、age、sex 这几个字段上都对可输入的数据的类型或长度加了限制，例如在 firstName 上的限制代码如下：

```
            <h:inputText id="firstName"
                                  value="#{UserRegistrationBean.firstName}"
                                  required="true">
                   <f:validateLength minimum="2" maximum="25" />
            </h:inputText>
```

也就是要求数据在 2～25 个字节之间。

在对 Email 的输入上还使用了管理 Bean 中的验证方法，代码如下：

```
                    <h:inputText id="email"
                                        value="#{UserRegistrationBean.email}"
validator="#{UserRegistrationBean.validateEmail}" >
                    </h:inputText>
```

如果用户的数据输入不是一个合法的 Email 地址，就会提示错误。

3．编写配置文件

在使用 JSF 开发中配置文件的编写也是比较繁琐的一件事，首先在 web.xml 文件中需要配置 JSF 的控制器，它是一个 Servlet：javax.faces.webapp.FacesServlet，在本应用中使用的 web.xml 文件的内容如下：

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
```

23.3　配置 Log4j

在应用程序代码中加入日志请求是需要大量的计划和精力的。观察显示，将近 4%的代码用于日志操作。因此，即使是一个中等大小的应用程序也会有上千行代码用于日志操作。在这种情况下，不手动修改实现对这些代码进行管理是很有意义的。

虽然 Log4j 环境是可以用程序配置的（例如 23.2.5 节的例子一样），但使用配置文件就更灵活了。配置文件可以被写成 XML 文件合适或者 Java Properties 格式（key＝value）。下面分别介绍这两种方式。

对 Log4j 环境的配置就是对 root logger 的配置，包括把 root logger 设置为哪个级别（Level）；为它设置相关联的 Appender 等。这些可以通过设置系统属性的方法来隐式地完成，也可以在程序中调用 XXXConfigurator.configure()方法来显式地完成。

Log4j 由 3 个重要的组件构成：日志信息的优先级、日志信息的输出目的地、日志信息的输出格式。在配置时它们也是主要的配置元素。

23.3.1　使用 Java Properties 配置

使用 Java Properties 配置就是按照 key＝value 的格式书写配置属性，使用这种方式是很容易理解的，而且也非常简单。

在使用这种方式进行配置时，任何 key 都必须以 Log4j 开头，其后跟相关的属性，下面介绍常用的几种日志配置方法。

1．配置根 Logger

配置根 Logger 的语法为：

```
log4j.rootLogger = [ level ] , appenderName, appenderName, …
```

其中，Level 是日志记录的优先级，分为 OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL（Level 类中预定义的）或者自己定义的级别。Log4j 建议只使用 4 个级别，优先级从高到低分别是 ERROR、WARN、INFO、DEBUG。通过在这里定义的级别，可以控制到应用程序中相应级别的日志信息的开关。比如在这里定义了 INFO 级别，则应用程序中所有 DEBUG 级别的日志信息将不被打印出来。appenderName 就是指定日志信息输出到哪个地方，可以同时指定多个输出目的地。

2．配置日志信息输出目的地 Appender

配置日志信息输出目的地 Appender 的语法为：

```
log4j.appender.appenderName = fully.qualified.name.of.appender.class
log4j.appender.appenderName.option1 = value1
……
log4j.appender.appenderName.option = valueN
```

```
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
### 配置日志输出的格式##
log4j.appender.A1.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} [%c]-[%p] %m%n

### 设置输出地 A2 到文件（文件大小到达指定尺寸时产生一个新的文件）##
log4j.appender.A2=org.apache.log4j.RollingFileAppender
### 文件位置##
log4j.appender.A2.File=E:/study/log4j/zhuwei.html
### 文件大小##
log4j.appender.A2.MaxFileSize=500KB
log4j.appender.A2.MaxBackupIndex=1
##指定采用 HTML 方式输出
log4j.appender.A2.layout=org.apache.log4j.HTMLLayout
```

5．其他输出地的配置

Log4j 的输出地可以是控制台、文件、回滚文件、发送日志邮件、数据库日志表和自定义标签等，不过不同的输出地的设置往往有些差别，下面举几个例子，分别实现输出到发送日志邮件、输出到数据库日志表、自定义标签，读者可以通过这些例子了解不同的输出地的配置流程。

```
###########################
#发送日志邮件
##########################
log4j.appender.MAIL=org.apache.log4j.net.SMTPAppender
log4j.appender.MAIL.Threshold=FATAL
#设置缓存大小
log4j.appender.MAIL.BufferSize=10
#设置邮件的发送者
log4j.appender.MAIL.From=chenyl@hollycrm.com
#设置邮件的接收者
log4j.appender.MAIL.SMTPHost=mail.hollycrm.com
#设置邮件主题
log4j.appender.MAIL.Subject=Log4J Message
#设置邮件的接收者
log4j.appender.MAIL.To=chenyl@hollycrm.com
#设置输出格式
log4j.appender.MAIL.layout=org.apache.log4j.PatternLayout
log4j.appender.MAIL.layout.ConversionPattern=[framework] %d - %c -%-4r [%t] %-5p %c %x -
%m%n

###########################
# 输出到数据库日志表
##########################
log4j.appender.DATABASE=org.apache.log4j.jdbc.JDBCAppender
#数据库的 URL，这里为 localhost 上的 test 数据库
log4j.appender.DATABASE.URL=jdbc:mysql://localhost:3306/test
#数据库驱动程序
log4j.appender.DATABASE.driver=com.mysql.jdbc.Driver
#数据库用户名
```

"info"、"warn"、"error"、"fatal"和"off"。当值为"off"时表示没有任何日志信息被输出。

- param [子元素，可有任意个]：创建 level 对象时传递给类构造方法的参数。

8. appender-ref 元素

appender-ref 元素引用一个 appender 元素的名字，为 logger 对象增加一个 appender。

- ref [必须设置的属性]：一个 appender 元素的名字的引用。
- appender-ref 元素没有子元素。

9. param 元素

param 元素在创建对象时为类的构造方法提供参数。它可以成为 appender、layout、filter、errorHandler、level、categoryFactory 和 root 等元素的子元素。

- name and value [#REQUIRED attributes]：提供参数的一组名值对。
- param 元素没有子元素。

10. 元素创建实例

（1）创建 FileAppender 对象。可以为 FileAppender 类的构造方法传递两个参数：File 表示日志文件名；Append 表示如文件已存在，是否把日志追加到文件尾部，可能取值为"true"和"false"（默认）。

```
<appender name="file.log" class="org.apache.log4j.FileAppender">
<param name="File" value="/tmp/log.txt" />
<param name="Append" value="false" />
<layout ... >
……
</layout>
</appender>
```

（2）创建 RollingFileAppender 对象。除了 File 和 Append 以外，还可以为 RollingFileAppender 类的构造方法传递两个参数：MaxBackupIndex 表示备份日志文件的个数（默认是 1 个），MaxFileSize 表示日志文件允许的最大字节数（默认是 10MB）。

```
<appender name="rollingFile.log" class="org.apache.log4j.RollingFileAppender">
<param name="File" value="/tmp/rollingLog.txt" />
<param name="Append" value="false" />
<param name="MaxBackupIndex" value="2" />
<param name="MaxFileSize" value="1024" />
<layout ... >
……
</layout>
</appender>
```

（3）创建 PatternLayout 对象。可以为 PatternLayout 类的构造方法传递参数 Conversion Pattern。

```
<layout class="org.apache.log4j.PatternLayout>
<param name="Conversion" value="%d [%t] %p - %m%n" />
</layout>
```

```
<logger name="shop.log">
<!-- 设置域名限制，即 shop.log 域及以下的日志均输出到下面对应的通道中 -->
    <level value="debug" />
    <!-- 设置级别 -->
    <appender-ref ref="cn.ac.ict.shop" /><!-- 与前面的通道 id 相对应 -->
</logger>

<root> <!-- 设置接收所有输出的通道 -->
    <appender-ref ref="cn.ac.ict.all" /><!-- 与前面的通道 id 相对应 -->
</root>

</log4j:configuration>
```

23.3.3　Log4j 配置实现过程

对 Log4j 环境的配置就是对 root logger 的配置，包括将 root logger 设置为哪个级别（level）；为它增加哪些 appender 等，这些可以通过设置系统属性的方法来隐式地完成，也可以在程序中调用 XXXConfigurator.configure()方法来显式地完成，所有其他的 logger 都是 root logger 的后代，所以它们（默认情况下）都将继承 root logger 的性质。

下面讲述默认的 log4j 初始化过程。

Logger 类的静态初始化块(static initialization block)中对 Log4j 的环境做默认的初始化。

注意： 如果程序员已经通过设置系统属性的方法来配置了 Log4j 环境，则不需要再显式地调用 XXXConfigurator.configure()方法来配置 Log4j 环境了。

Logger 的静态初始化块在完成初始化过程时将检查一系列 Log4j 定义的系统属性。它所做的事情如下：

- 检查系统属性 log4j.defaultInitOverride，如果该属性被设置为 false，则执行初始化；否则（只要不是 false，无论是什么值，甚至没有值，都是否则），跳过初始化。
- 把系统属性 log4j.configuration 的值赋给变量 resource。如果该系统变量没有被定义，则把 resource 赋值为"log4j.properties"。
- 试图把 resource 变量转化成为一个 URL 对象 url。如果一般的转化方法行不通，就调用 org.apache.log4j.helpers.Loader.getResource(resource, Logger.class)方法来完成转化。
- 如果 url 以".html"或".xml"结尾，则调用方法 DOMConfigurator.configure(url)来完成初始化；否则调用方法 PropertyConfigurator.configure(url)来完成初始化。如果 url 指定的资源不能被获得，则跳出初始化过程。

注意： 在 Apache 的 Log4j 文档中建议使用定义 log4j.configuration 系统属性的方法来设置默认的初始化文件是一个好方法。

所以，对于使用 Java Properties 配置文件，要使用 PropertyConfigurator.configure(url)来完成初始化，而对于使用 xml 的配置文件需要使用 DOMConfigurator.configure(url)来完成初始化。

23.4　Web 应用中使用 Log4j 的例子

在本书“JSP 与 Java Mail Web 应用”一章中介绍的实例中就是使用 Log4j 进行日志记录的，下面介绍其使用的配置文件：

```
#ConfigLog_ln.properties
#设置 logger 和 level
log4j.rootCategory=DEBUG, R
#输出到文件
log4j.appender.R=org.apache.log4j.FileAppender
#输出的日志文件名（在项目的根目录中存放）
log4j.appender.R.File=D:\\Tomcat 5.0\\logs\\JMailLog4j.txt
#文件格式为自定义模式(共有 4 种可选)
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%m%n
```

编写好配置文件后，还需要在程序中配置一下，让程序知道配置文件的位置，并创建合适的日志进行记录。配置 Log4j 的代码（Log4j.java）如下：

```
package cn.ac.ict.JavaMail;

import java.net.URL;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class Log4j {
          public static Logger
     logger = Logger.getLogger(cn.ac.ict.JavaMail.Log4j.class.getName());

//调用配置文件
public static void ConfigLog() {
   String resource = "ConfigLog.properties";
   URL configFileResource =
       cn.ac.ict.JavaMail.Log4j.class.getResource(resource);
   PropertyConfigurator.configure(configFileResource);
}
//调用配置文件二（日志换行输出）
public static void ConfigLog_ln() {
   String resource = "ConfigLog_ln.properties";
   URL configFileResource =
       cn.ac.ict.JavaMail.Log4j.class.getResource(resource);
   PropertyConfigurator.configure(configFileResource);
}
}
```

在这个类中使用的方法都是 static 的，可以直接调用来配置 Log4j，配置好后，就可以

在程序中使用了。

例如，在 MailUserInfoBean 类的构造函数中调用了 ConfigLog_ln()对其进行配置：

```
        public MailUserInfoBean(){
                Log4j.ConfigLog_ln();

        }
```

在程序中就可以这样使用 Log4j 进行日志的记录了：

```
……
catch (NoSuchProviderException e) {
                        e.printStackTrace();
                        Log4j.logger.debug("Get a Store error!");
                        Log4j.logger.debug(e);
                        return JMailUtil.FAILED;
                } catch (MessagingException e) {
                        Log4j.logger.debug("Get a Store error!");
                        e.printStackTrace();
                        Log4j.logger.debug(e);
                        return JMailUtil.FAILED;
                }
……
```

程序运行后，日志的内容大致如下：

```
The Store is connected!
邮件被成功删除!
邮件被成功删除!
邮件被成功删除!
发送邮件失败!javax.mail.SendFailedException: No recipient addresses
发送邮件成功!
发送邮件成功!
```

23.5　小　　结

Log4j 是一个优秀的日志记录工具，通过使用 Log4j 可以很方便地进行程序的调试以及程序流程的跟踪和分析。

在本章中首先介绍了一个在 JSP 文件中使用日志的例子，然后介绍了很多相关的基础知识，第一个例子是在程序中配置 Log4j 的，这样做很不方便，因此在后面一个 Java Mail 的应用中将其改为使用配置文件的方式，大大简化了日志的记录，读者可以尝试使用这种方式进行日志记录操作。

第 24 章　使用 XDoclet 简化 JSP 的 Web 开发

XDoclet 是一个用于简化配置文件书写的开源工具，使用它可以大大减少开发 Java 程序所需要书写配置文件的时间。在本章中介绍如何使用 XDoclet 简化 JSP 的 Web 开发。

24.1　快速体验 XDoclet——使用 XDoclet 的简单例子

24.1.1　XDoclet 介绍

从本书前面的介绍可以看到，使用 JSP 技术进行开发，需要很多的配置工作，往往要发布一个组件需要写几个 XML 文件，而任何一个文件的某个地方出了错误都会造成配置失败，调试起来也非常麻烦，为了解决这个难题，本章介绍一个非常好用的用于生成配置文件的开源工具 XDoclet。

XDoclet 是一个通用的代码生成实用程序，是一个扩展的 Javadoc Doclet 引擎（现已与 Javadoc Doclet 独立），XDoclet 是 EJBDoclet 的后继者，而 EJBDoclet 是由 Rickard Oberg 发起的（XDoclet 的官方主页：http://xdoclet.sourceforge.net/xdoclet/index.html）。

XDoclet 因为可以自动生成 EJB 复杂的接口和部署描述文件而使得很多人知道它，但现在的 XDoclet 已经发展成了一个全功能的、面向属性的代码生成框架。J2EE 代码生成只是 XDoclet 的一个应用方面，它可以完成的任务已经远远超越了 J2EE 和项目文档的生成。

在下面会介绍一个使用 XDoclet 的简单例子，使用 XDoclet 为 Servlet 生成配置信息。

24.1.2　安装配置 XDoclet

XDoclet 是开源的免费软件，可以从其官方主页上免费下载（目前最新版本为 1.2.3，xdoclet-bin-1.2.3.zip），将下载的压缩文件解压到某个目录，并把这个目录称为 XDOCLET_HOME，在使用 XDoclet 时一般要结合 ANT 使用，可以在 build 文件中设置使用的 XDOCLET_HOME，所以这里不多作介绍。

24.1.3　Java 源程序和添加注释

下面编写一个简单的用户登录页面，这个 Servlet 可以根据配置信息验证用户的身份，

除了像开发其他 Java 程序要写的代码和注释外，还需要添加 XDoclet 需要的额外注释，XDoclet 就是根据这些注释（以@tags 的格式出现）来生成配置文件的。下面是 Java 源程序：

```
package cn.ac.ict.XDoclet;

import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * @web.servlet name="LoginServlet" display-name="Login Servlet XDoclet Sample"
 *                      load-on-startup="1"
 * @web.servlet-init-param name="username"
 *                                      value="${servlet.username}"
 * @web.servlet-init-param name="password"
 *                                      value="${servlet.password}"
 * @web.servlet-mapping url-pattern="/Login/*"
 */
public class XDocletLoginSample extends HttpServlet {

        private String username;
        private String password;

        public void init(ServletConfig config) throws ServletException {
                //从 web.xml 中获得初始化参数
                super.init(config);
                this.username = config.getInitParameter("username");
                this.password = config.getInitParameter("password");
        }

        protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {
                doPost(request, response);
        }

        protected void doPost(HttpServletRequest request,
                        HttpServletResponse response) throws ServletException, IOException {
              //首先设置文档类型
              response.setContentType("text/html; charset=GBK");
              //获取输出流
              java.io.PrintWriter out = response.getWriter();

              out.write("<!DOCTYPE  HTML  PUBLIC  \"-//W3C//DTD  HTML  4.01  Transitional//EN\"
```

```
out.write("\r\n");
out.write("    <tr>\r\n");
out.write("      <td align=\"center\"><span class=\"style3\">用户名或者密码错误！</span></td>\r\n");
out.write("    </tr>\r\n");
out.write("    ");
     }
       if(!login){
out.write("\r\n");
out.write("    <tr>\r\n");
out.write("      <td   align=\"center\">用户名：<input name=\"username\" type=\"text\"></td>\r\n");
out.write("    </tr>\r\n");
out.write("    <tr>\r\n");
out.write("      <td align=\"center\">密 码：\r\n");
out.write("        <input name=\"password\" type=\"text\"></td>\r\n");
out.write("    </tr>\r\n");
out.write("    <tr>\r\n");
out.write("          <td   align=\"center\"><input   name=\"submit\"   type=\"submit\"   value=\" 提 交
\"> <input name=\"reset\" type=\"reset\" value=\"重置\"></td>\r\n");
out.write("    </tr>\r\n");
out.write("    ");
   }else{
out.write("\r\n");
out.write("     <tr>\r\n");
out.write("      <td align=\"center\"><span class=\"style3\">您已成功登录！ </span></td>\r\n");
out.write("    </tr>\r\n");
out.write(" \r\n");
out.write("    ");
   }
out.write("\r\n");
out.write("</table>\r\n");
out.write("</form>\r\n");
out.write("\r\n");
out.write("</body>\r\n");
out.write("</html>\r\n");

      }
}
```

24.1.4 书写 build.xml 文件

本例是将 XDoclet 和 Ant 配合使用的，下面是编译和部署这个简单应用时 Ant 所需要的 build.xml 文件：

```
<?xml version="1.0" encoding="GBK"?>
```

```
<project name="filtering" default="deploy" basedir=".">
 <description>一个简单的 XDoclet 实例</description>

 <!-- 载入属性文件 -->
 <property file="build.properties"/>

 <!-- 定义类路径 -->
 <path id="web.classpath">
  <pathelement location="${tomcat.home}/common/lib/servlet-api.jar"/>
  <pathelement location="${tomcat.home}/common/lib/jsp-api.jar"/>
 </path>
 <path id="xdoclet.classpath">
  <fileset dir="${xdoclet.home}/lib">
   <include name="*.jar"/>
  </fileset>
  <path refid="web.classpath"/>
 </path>

 <!-- 初始化,建立目录 -->
 <target name="init">
  <mkdir dir="${dist.dir}"/>
  <mkdir dir="${dist.dir}/WEB-INF"/>
  <mkdir dir="${dist.dir}/WEB-INF/classes"/>
 </target>

 <!-- XDoclet 的 WebDoclet 任务 -->
 <target name="webdoclet" depends="init">
  <taskdef
      name="webdoclet"
      classpathref="xdoclet.classpath"
      classname="xdoclet.modules.web.WebDocletTask"/>

  <webdoclet destDir="${dist.dir}/WEB-INF" force="${xdoclet.force}">
      <deploymentdescriptor Servletspec="2.4" xmlencoding="GBK"/>
   <fileset dir="${src.dir}" includes="**/*.java"/>
  </webdoclet>
 </target>

 <!-- 编译与部署 -->
 <target name="deploy" depends="webdoclet">
  <javac srcdir="${src.dir}" destdir="${dist.dir}/WEB-INF/classes">
   <classpath refid="web.classpath"/>
  </javac>
  <jar destfile="${tomcat.home}/webapps/${app.name}.war" basedir="${dist.dir}"/>
 </target>
</project>
```

其中最主要的就是使用了 XDoclet 的 WebDoclet 任务，它是 XDoclet 预定义好的，在使用时需要使用<taskdef>元素引入这个任务，它的实现类是 xdoclet.modules.web.WebDocletTask。在上面的 build 文件中包含了一个属性配置文件 build.properties，它的内容要根据 Tomcat 和 XDoclet 安装的情况进行修改，大致内容如下：

```
################ 环境设置 ###################

#  Tomcat 安装主目录
tomcat.home=D:/Tomcat 5.0
#  xdoclet 安装目录
xdoclet.home=E:/xdoclet-1.2.3
#  web 的临时目录
dist.dir=./dist
#  源文件目录
src.dir=./src
#  发布的程序名
app.name=XDocletLogin
#  Servlet 参数，可以改变
servlet.username=XDocletUser
servlet.password=XDocletPass
```

在上面的文件中设置了合法的用户名和密码，可以作为 Servlet 的初始化参数，在 Servlet 的注释中已有体现。

24.1.5　运行实例

按照上面的安排准备好所有的文件后，这个应用的目录结构如图 24.1 所示。这时在这个目录下运行 Ant 命令，效果如图 24.2 所示。

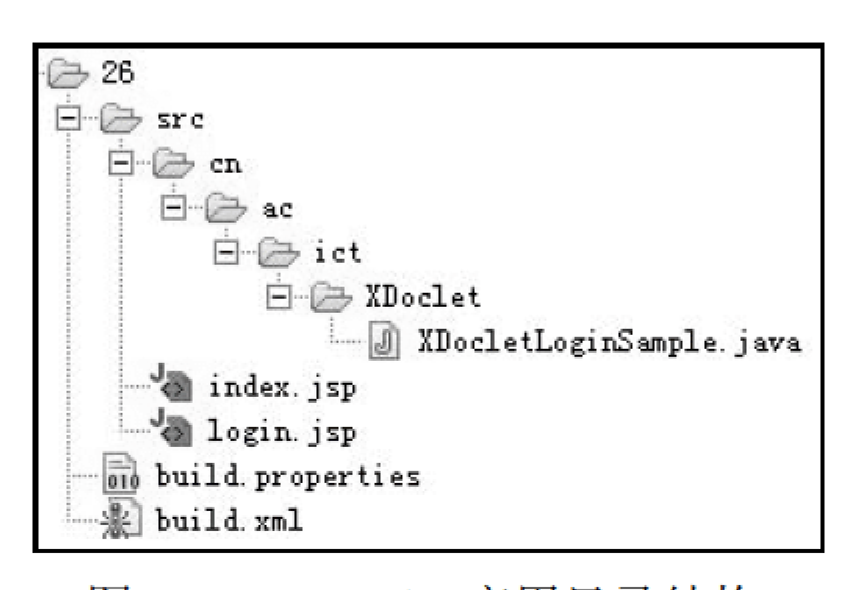


图 24.1　XDoclet 应用目录结构

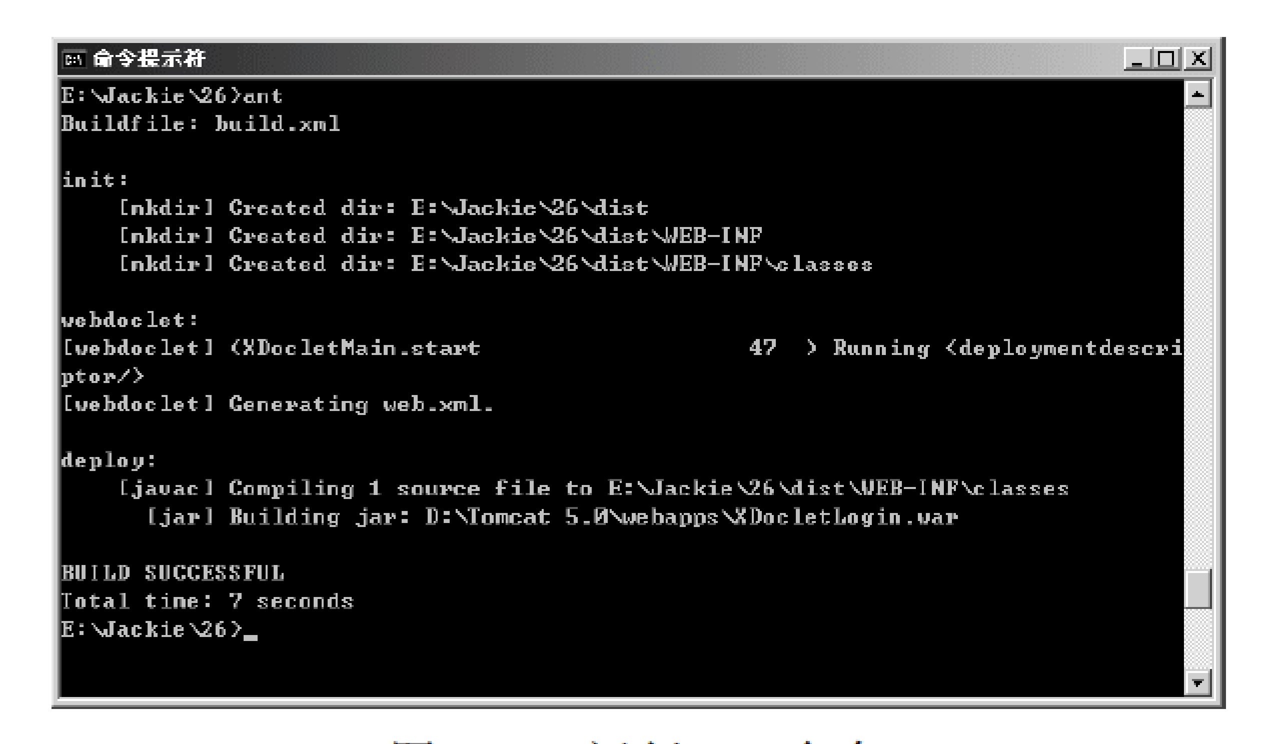


图 24.2　运行 Ant 命令

启动 Tomcat，然后在浏览器地址栏中输入如下地址：http://localhost:8080/XDocletLogin/Login，任意输入用户名和密码后，可能的效果图如图 24.3 所示。

输入正确的用户名和密码：用户名是 XDocletUser，密码是 XDocletPass，可以看到页面效果如图 24.4 所示。

```
  <servlet>
      <display-name>Login Servlet XDoclet Sample</display-name>
      <servlet-name>LoginServlet</servlet-name>
      <servlet-class>cn.ac.ict.XDoclet.XDocletLoginSample</servlet-class>

      <init-param>
          <param-name>username</param-name>
          <param-value>XDocletUser</param-value>
      </init-param>
      <init-param>
          <param-name>password</param-name>
          <param-value>XDocletPass</param-value>
      </init-param>

      <load-on-startup>1</load-on-startup>

  </servlet>

 <!--
 To use non XDoclet servlets, create a servlets.xml file that
 contains the additional servlets (eg Struts) and place it in your
 project's merge dir.   Don't include servlet-mappings in this file,
 include them in a file called servlet-mappings.xml and put that in
 the same directory.
 -->

  <servlet-mapping>
      <servlet-name>LoginServlet</servlet-name>
      <url-pattern>/Login/*</url-pattern>
  </servlet-mapping>

  <!--
  To specify mime mappings, create a file named mime-mappings.xml, put it in your project's
mergedir.
  Organize mime-mappings.xml following this DTD slice:

  <!ELEMENT mime-mapping (extension, mime-type)>
  -->

  <!--
  To specify error pages, create a file named error-pages.xml, put it in your project's mergedir.
  Organize error-pages.xml following this DTD slice:

  <!ELEMENT error-page ((error-code | exception-type), location)>
  -->
```

属性的最后一行。

下面来看看 Servlet 文件中的注释都起了什么作用，在 Servlet 文件中有如下注释：

```
/**
 * @web.servlet name="LoginServlet" display-name="Login Servlet XDoclet Sample"
 *                    load-on-startup="1"
 * @web.servlet-init-param name="username"
 *                                   value="${servlet.username}"
 * @web.servlet-init-param name="password"
 *                                   value="${servlet.password}"
 * @web.servlet-mapping url-pattern="/Login/*"
 */
```

首先@web.servlet name 标签指定了这个 Servlet 在 web.xml 中配置时使用的名字（<servlet-name>元素的值），display-name 用于生成<display-name>元素，load-on-startup 用于生成<load-on-startup>元素，也就是说通过第一个标签生成了如下的配置信息：

```
    <display-name>Login Servlet XDoclet Sample</display-name>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>cn.ac.ict.XDoclet.XDocletLoginSample</servlet-class>
    <load-on-startup>1</load-on-startup>
```

然后是@web.servlet-init-param 标签，它指定了 Servlet 的初始化参数，用于生成<init-param>元素，其中 name 属性生成子元素<param-name>、value 属性生成子元素<param-value>。也就是说，这个标签最终生成的配置信息是：

```
    <init-param>
        <param-name>username</param-name>
        <param-value>XDocletUser</param-value>
    </init-param>
    <init-param>
        <param-name>password</param-name>
        <param-value>XDocletPass</param-value>
    </init-param>
```

最后是@web.servlet-mapping 标签，它指定了这个 Servlet 的映射信息，url-pattern 指定了映射到这个 Servlet 的 URL，这个标签生成了如下的配置信息：

```
  <servlet-mapping>
      <servlet-name>LoginServlet</servlet-name>
      <url-pattern>/Login/*</url-pattern>
  </servlet-mapping>
```

其中<servlet-name>元素由@web.servlet name 标签指定。

这些就是配置文件的形成来源，XDoclet 能够生成配置文件就是根据这些注释来实现的，如果注释写得不正确或者不规范就无法生成想要的信息。

24.3　使用 XDoclet 进行 Web 开发

使用 XDoclet 可以为开发 EJB、Servlet、Filter 等提供很大的方便，使用 XDoclet 进行 Web 开发更是可以省去很多工作，而且根据上面的介绍，读者也可以了解到使用 XDoclet 关键要用合适的标签写好注释，下面就结合 XDoclet 为 Web 开发提供的标签介绍如何使用 XDoclet 加速 Web 开发的进程。

24.3.1　开发 Struts

（1）@struts 标签用于为开发 Struts 提供支持，其中它的类级别的标签如表 24.1 所示。

表 24.1　@struts的类级别的标签

标　　签	描　　述
@struts.action	用于定义Action类和它的属性
@struts.action-exception	定义Action类的异常处理器
@struts.action-forward	为Action类定义局部转发
@struts.action-set-property	为Action类创建set-property
@struts.dynaform	定义一个动态的Form Bean和它的属性
@struts.form	定义一个Form Bean和它的属性

（2）@struts.action 用于定义 Action 类和它的属性，它的参数及其描述如表 24.2 所示。

表 24.2　@struts.action标签的参数及其描述

参　　数	类　　型	描　　述	是 否 必 需
name	text	Action的名字，在这个Struts应用中是惟一的	是
type	text	为这个Action实例化的类，默认就是当前类	否
className	text	用于为这个Action提供服务的ActionMapping的子类的全名	否
path	text	这个Action符号的路径	是
scope	text	定义Action的范围，为request、session、application的一种	是
input	text	为这个Action提供输入的路径	是
roles	text	允许访问这个ActionMapping对象的安全角色名，用逗号分隔	否
validate	text	这个Action的验证标志，默认为true	是
parameter	text	这个Action的可选参数	是

例如下面的一段注释：

```
@struts.action
      path="/struts/foo"
```

将会生成如下的配置信息：

```
     <action
       path="/struts/foo"
```

```
      validate="true"
   >
      <forward
         name="success"
         path="/struts/getAll.do"
         redirect="false"
      />
   </action>
```

注意： 如何使用上面的例子将在本章最后一小节介绍。

24.3.2　开发 Servlet 过滤器

（1）@web.filter 标签用于定义 Servlet 过滤器，它只能对 Servlet 加注释。它的参数及其描述如表 24.6 所示。

表 24.6　@web.filter标签的参数及其描述

参　　数	类　　型	描　　述	是 否 必 需
name	text	过滤器的名字，在当前Web应用中是惟一的	是
display-name	text	过滤器的显示名称	否
icon	text	过滤器的图标	否
description	text	描述信息	否

（2）@web.filter-init-param 用于为过滤器指定初始化参数，它的参数及其描述如表 24.7 所示。

表 24.7　@web.filter-init-param标签的参数及其描述

参　　数	类　　型	描　　述	是 否 必 需
name	text	参数的名字	是
value	text	参数的值	否
description	text	描述信息	否

（3）@web.filter-mapping 标签用于为过滤器定义映射信息，它的参数及其描述如表 24.8 所示。

表 24.8　@web.filter-mapping标签的参数及其描述

参　　数	类　　型	描　　述	是 否 必 需
url-pattern	text	过滤器符合的URL	否
servlet-name	text	过滤器Servlet的名字	否
dispatcher	text	与这个过滤器有关的请求转发器	否

例如下面的一个过滤器的例子用到了关于过滤器定义的标签，下面是 TimerFilter.java 的源代码：

```
package cn.ac.ict;
```

```
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
/* @web.filter
 *       display-name="Timer Filter"
 *       name="TimerFilter"
 *
 * @web.filter-init-param
 *       name="param1"
 *       value="value1"
 *
 * @web.filter-init-param
 *       name="param2"
 *       value="value2"
 *
 * @web.filter-mapping
 *       url-pattern="*.xml"
 */
public class TimerFilter implements Filter {

    private FilterConfig config = null;

    public void init(FilterConfig config) throws ServletException {
        this.config = config;
    }

    public void destroy() {
        config = null;
    }

    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException,
ServletException {
        long before = System.currentTimeMillis();

        chain.doFilter(request, response);

        long after = System.currentTimeMillis();

        String name = " ";
        if (request instanceof HttpServletRequest)
            name = ((HttpServletRequest) request).getRequestURI();

        config.getServletContext().log(name + ": " + (after - before) + "ms");
    }
}
```

这个文件被 XDoclet 解析后会为这个 Filter 生成如下的配置信息：

```
<filter>
    <filter-name>TimerFilter</filter-name>
    <display-name>Timer Filter</display-name>
    <filter-class> cn.ac.ict.TimerFilter</filter-class>
    <init-param>
        <param-name>param1</param-name>
        <param-value>value1</param-value>
    </init-param>
    <init-param>
        <param-name>param2</param-name>
        <param-value>value2</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>TimerFilter</filter-name>
    <url-pattern>*.xml</url-pattern>
</filter-mapping>
```

注意：如何使用上面的例子将在本章最后一小节介绍。

24.3.3 开发自定义标签

@jsp.tag 标签用于给自定义标签加注释，它的参数及其描述如表 24.9 所示。

表 24.9 @jsp.tag标签的参数及其描述

参 数	类 型	描 述	是否必需
name	text	JSP标签的名字	是
tei-class	text	JSP的tei类名	否
body-content	text	JSP标签体内容：tagdependent、JSP或empty，其默认值是JSP	否
display-name	text	标签的显示名称	否
small-icon	text	标签的小图标	否
large-icon	text	标签的大图标	否
description	text	标签的描述信息	否

24.3.4 运行例子

在 24.3.1 节和 24.3.2 节中介绍了两个例子，下面介绍如何演示这两个例子，首先要编写一个 build.xml 文件，其内容如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="XDoclet Examples" default="compile" basedir=".">
    <property file="build-dist.properties"/>
```

```
    <path id="samples.class.path">
        <fileset dir="${lib.dir}">
            <include name="*.jar"/>
        </fileset>
        <fileset dir="${samples.lib.dir}">
            <include name="*.jar"/>
        </fileset>
        <fileset dir="${dist.lib.dir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="init">
        <tstamp>
            <format property="TODAY" pattern="d-MM-yy"/>
        </tstamp>

        <taskdef
            name="webdoclet"
            classname="xdoclet.modules.web.WebDocletTask"
            classpathref="samples.class.path"
            />
    </target>
   <!--   建立临时目录 -->
    <target name="prepare" depends="init">
        <mkdir dir="${samples.classes.dir}"/>
        <mkdir dir="${samples.gen-src.dir}"/>
        <mkdir dir="${samples.meta-inf.dir}"/>
    </target>
   <!--   webdoclet 目标 -->
    <target name="webdoclet" depends="prepare"
description="Generate deployment descriptors (run actionform to generate forms first)">

        <webdoclet
            destdir="${samples.gen-src.dir}"
            mergedir="parent-fake-to-debug"
            excludedtags="@version,@author,@todo"
            addedtags="@xdoclet-generated at ${TODAY},@copyright The XDoclet Team, @author
XDoclet,@version ${version}"
            force="${samples.xdoclet.force}"
            verbose="false"
            >
            <fileset dir="${samples.java.dir}">
                <include name="**/*Servlet.java"/>
                <include name="**/*Filter.java"/>
```

这个文件中的关键部分就是 WebDoclet 目标。

在初始化目标中定义了名为 WebDoclet 的任务，在 WebDoclet 目标调用了这个任务，并使用了两个标签<strutsconfigxml>和<deploymentdescriptor>，分别用于生成 web.xml 文件和 struts-config.xml 文件，还调用了<strutsvalidationxml>来生成 Struts 的验证信息。

在本代码开头包含了一个文件，这个文件中声明了很多需要用到的属性，其代码如下：

```
xdoclet.root.dir=E:/xdoclet-1.2.3
lib.dir = ${xdoclet.root.dir}/lib
dist.lib.dir = ${lib.dir}

samples.dir = .
samples.dist.dir = ${samples.dir}/target
samples.lib.dir = ${samples.dir}/lib
samples.src.dir = ${samples.dir}/src
samples.java.dir = ${samples.src.dir}/java
samples.gen-src.dir = ${samples.dist.dir}/gen-src

samples.meta-inf.dir = ${samples.dist.dir}/meta-inf
samples.web-inf.dir = ${samples.dist.dir}/web-inf
samples.merge.dir = ${samples.src.dir}/merge
samples.classes.dir = ${samples.dist.dir}/classes
samples.web.dir = ${samples.src.dir}/web
samples.xdoclet.force = false
```

读者在运行这个程序时需要把 XDoclet 的根路径修改一下。各个文件都改写完成后，其整个文件夹的结构如图 24.5 所示。

在该目录下运行 Ant 命令可以看到各种信息，运行结束后可以在其生成的子目录 target 下查找相应的配置文件。

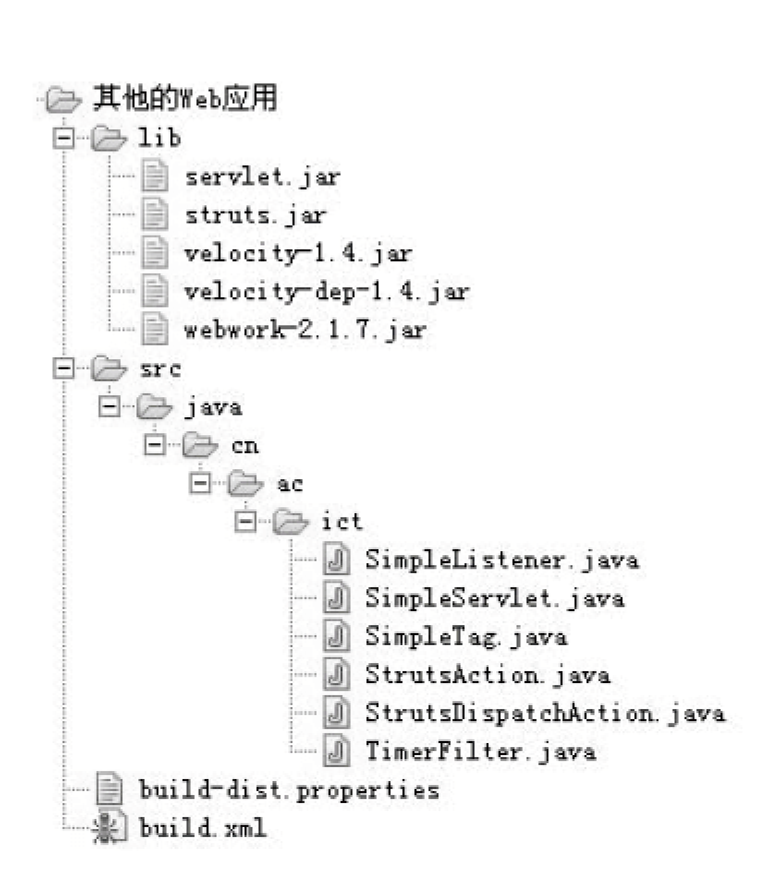


图 24.5　程序结构

24.4　小　　结

XDoclet 是一个优秀的通用的代码生成实用程序，使用它简化了很多 Web 应用的开发。在本章中只是介绍了它的一小部分应用，它还可以结合 Hibernate 等工具开发。通过前面的介绍可以看到，如果能够正确地书写需要的配置注释，使用 XDoclet 还是很方便的。

```
C:\WINDOWS\System32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\zhw>ant
Buildfile: build.xml does not exist!
Build failed
C:\Documents and Settings\zhw>ant build.xml
Buildfile: build.xml does not exist!
Build failed
C:\Documents and Settings\zhw>_
```

图 25.1　Ant 安装成功测试

Ant 安装完成后，可以看到如下的目录布局：

```
ant
    +--- bin   //包括各种二进制启动命令
    |
    +--- lib   // 包括 ant 运行需要的包文件
    |
    +--- docs //包括这种文档
    |        +--- ant2      // ant2 运行需要的条件的一个简单描述
    |        |
    |        +--- images   // HTML 文档中的各种图片
    |        |
    |        +--- manual   // Ant  文档
    |
    +--- etc //  包括具有各种功能的 xsl 文件
```

25.1.3　编写应用类文件

在这里编写一个 JavaBeans 文件和 JSP 文件，然后使用 Ant 编译 JavaBeans，并把编译后的字节码文件复制到某个 Web 应用的合适目录下，使用的 JavaBeans 文件代码如下：

```
package cn.ac.ict;

import java.io.Serializable;
import java.util.Date ;

public class Product implements Serializable{
//JavaBeans 的属性
      private String pname;
      private String pcomp;
      private Date pmadeyear;
      private float price;
      private int amount;

      public Product(){

      }
//JavaBeans 属性的获取和设置方法
      public String getPname(){
```

```
			return pname;
		}
		public String getPcomp(){
			return pcomp;
		}

		public Date getPmadeyear(){
			return pmadeyear;
		}
		public float getPrice(){
			return price;
		}

		public int getAmount(){
			return amount;
		}

		public void setPname(String productname){
			pname = productname;
		}
		public void setPcomp(String productcomp){
			pcomp = productcomp;
		}

		public void setPmadeyear(Date madeyear){
			pmadeyear = madeyear;
		}
		public void setPrice(float price){
			this.price = price;
		}

		public void setAmount(int pamount){
			amount = pamount;
		}

}
```

使用的 JSP 文件的代码如下：

```
<%@ page language="java" pageEncoding="GB2312" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
  <head>
    <title>添加商品 JSP 页面</title>
  </head>
<body bgcolor="#FFFFFF">
<jsp:useBean id="product" scope="request" class="cn.ac.ict.Product" />
<jsp:setProperty name="product" property="*"   />

<form action="addproduct.jsp" method="get">
```

```
<table width="580" border="0" cellspacing="0" cellpadding="0" align="center">
<thead>添加新的商品</thead>
  <tr>
    <td>商品名称</td>
    <td><input name="pname" type="text"></td>
  </tr>
  <tr>
    <td>生产商家</td>
    <td><input name="pcomp" type="text"></td>
  </tr>
  <tr>
    <td>生产日期</td>
    <td><input name="pmadeyear" type="text"></td>
  </tr>
  <tr>
    <td>价格</td>
    <td><input name="price" type="text"></td>
  </tr>
  <tr>
    <td>数量</td>
    <td><input name="amount" type="text"></td>
  </tr>
  <tr>
    <td><input name="submit" type="button" value="提交"></td>
    <td><input name="reset" type="button" value="重置"></td>
  </tr>
</table>
</form>

</body>
</html>
```

25.1.4　编写相关的 build.xml 文件

准备好了需要使用的 Java 文件和 JSP 文件后，就可以编写运行 Ant 需要使用的 build.xml 文件了，下面是 build.xml 文件的完整代码：

```
<!--     定义的 Project 名称、默认执行的 target 是 dist      -->
<project name="SimpleProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
<!--     定义全局属性      -->
  <property name="src" location="src"/>
  <property name="dist"   location="dist"/>

<!--名为 init 的 target，它完成建立临时目录的工作，并把需要使用的源文件复制到合适位置-->
  <target name="init">
     <mkdir dir="${dist}"/>
```

```
        <mkdir dir="${dist}/WEB-INF"/>
        <mkdir dir="${dist}/WEB-INF/classes"/>
        <copy todir="${dist}" >
            <fileset dir="${src}">
                <include name="*.jsp" />
                <exclude name="build.xml" />
            </fileset>
        </copy>
        <copy todir="${dist}/WEB-INF" >
            <fileset dir="${src}">
                <include name="web.xml" />
                <exclude name="build.xml" />
            </fileset>
        </copy>
    </target>

<!--      名为 compile 的 target，它编译 JavaBeans 文件      -->
    <target name="compile" depends="init"
            description="compile the source " >
        <javac srcdir="${src}" destdir="${dist}/WEB-INF/classes"/>
    </target>

<!--      名为 dist 的 target，它将 Web 应用打包      -->
    <target name="dist" depends="compile"
            description="generate the distribution" >
      <jar destfile="${basedir}/FirstAnt.war" basedir="${dist}"/>
    </target>

      <!-- 删除编译和发布时使用的临时目录 -->
    <target name="clean"
            description="clean up" >
      <delete dir="${dist}"/>
    </target>
</project>
```

25.1.5　使用 Ant 运行

所有的文件都准备好后，这个 Web 应用的目录结构如图 25.2 所示。在该目录下运行 Ant 命令，可以看到命令行提示效果如图 25.3 所示。

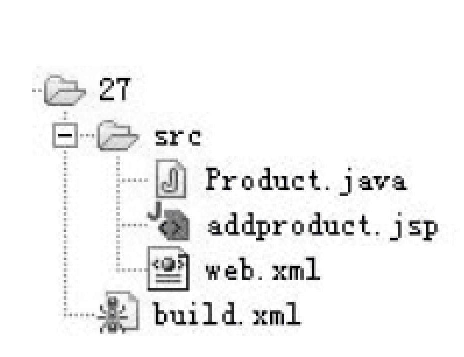


图 25.2　Web 应用的目录结构

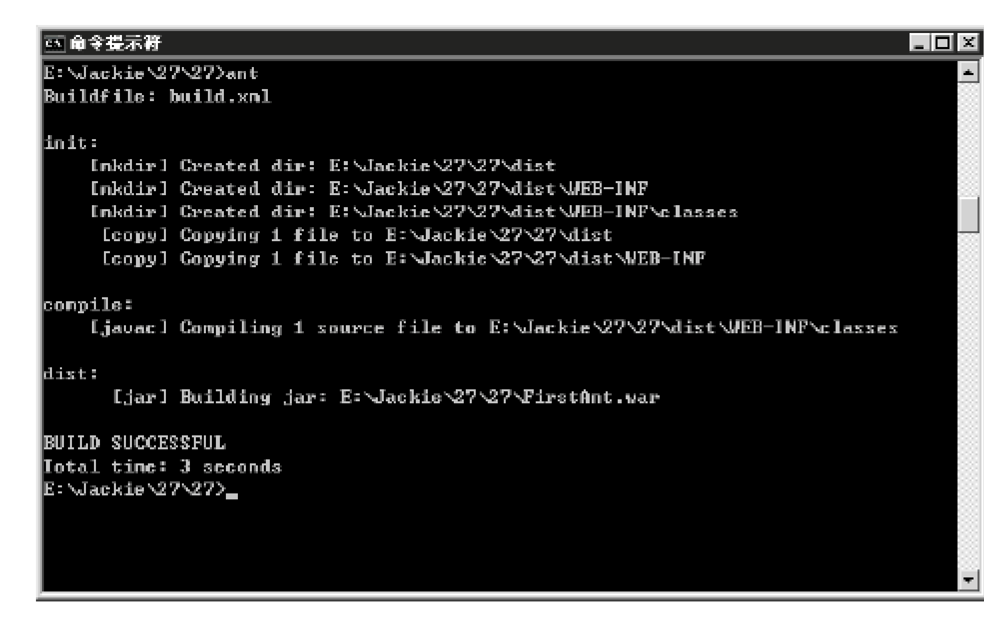


图 25.3　命令行提示效果

25.2.2　target 元素介绍

一个项目可以定义一个或多个 target，一个 target 是一系列想要执行的任务的集合。执行 Ant 时，可以选择执行哪个 target。当没有给定 target 时，使用 project 的 default 属性所确定的 target。

一个 target 可以依赖于其他的 target。例如，可能会有一个 target 用于编译程序，一个 target 用于生成可执行文件。在生成可执行文件之前必须先编译通过，所以生成可执行文件的 target 依赖于编译 target。Ant 会自动处理这种依赖关系。

然而，Ant 的 depends 属性只指定了 target 应该被执行的顺序，如果被依赖的 target 无法运行，这种 depends 对于指定了依赖关系的 target 就没有影响。

Ant 会依照 depends 属性中 target 出现的顺序（从左到右）依次执行每个 target。然而，要记住的是只要某个 target 依赖于一个 target，后者就会被先执行。

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D" depends="C,B,A"/>
```

假定要执行 target D。从它的依赖属性来看，可能认为先执行 C，然后 B，最后 A 被执行。不过，由于 C 依赖于 B，B 依赖于 A，所以先执行 A，然后 B，然后 C，最后 D 被执行。

target 元素支持的属性包括 name、depends、if、unless 和 description，如表 25.2 所示。

表 25.2　target元素的属性描述

属　性　名	描　　述	是 否 必 需
name	target的名字	是
depends	依赖表，用逗号分隔的target的名字列表	否
if	执行target需要设定的属性名	否
unless	执行target需要清除设定的属性名	否
description	关于target功能的简短描述	否

如果（或如果不）某些属性被设定才执行某个 target。这样，允许根据系统的状态（Java version、OS、命令行属性定义等）来更好地控制 build 的过程。要想让一个 target 这样做，应该在 target 元素中加入 if（或 unless）属性，以及 target 应该有所判断的属性。例如：

```
<target name="build-module-A" if="module-A-present"/>
<target name="build-own-fake-module-A" unless="module-A-present"/>
```

如果没有 if 或 unless 属性，target 总会被执行。

25.2.3　task 元素介绍

一个 task 是一段可执行的代码。一个 task 可以有多个属性。属性只可能包含对 property 的引用。这些引用会在 task 执行前被解析。

这个例子是用于编译 Java 源程序的，其中 srcdir 指定源文件所在的目录；destdir 指定编译后的 class 文件存放的目录；includes 表示只有 mypackage/p1 和 mypackage/p2 目录下的文件必须被包含；而 excludes 表示 mypackage/p1/testpackage/目录下的文件被排除在外，classpath 指定了编译这个文件使用的类路径；debug 表示是否有调试信息。

```
<jar destfile="${dist}/lib/app.jar"
       basedir="${build}/classes"
       excludes="**/Test.class"
  />
```

这个例子是用于把文件打包的，其中 destfile 指定了打包后包的文件名；basedir 指定被包含文件的基路径；而 excludes 则表示任何目录下的 Test.class 文件都不被打包。

```
       <java classname="test.Main">
         <arg value="-h"/>
         <classpath>
           <pathelement location="dist/test.jar"/>
           <pathelement path="${java.class.path}"/>
         </classpath>
       </java>
```

这个例子用于执行 Java 程序，classname 指定了被执行的类的完整类名；<arg>定义了传给这个类的参数；<classpath>元素定义了运行需要的类路径信息。

25.2.6　build.xml 实例分析

在 25.1 节中介绍了一个使用 Ant 的实际例子，现在对其使用的 build.xml 文件进行分析，以便更好地理解如何编写 build.xml 文件。

```
<!--     定义的 project 名称、默认执行的 target 是 dist     -->
<project name="SimpleProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
<!--     定义全局属性     -->
  <property name="src" location="src"/>
  <property name="dist"   location="dist"/>

<!--名为 init 的 target，它完成建立临时目录的工作，并把需要使用的源文件复制到合适位置-->
  <target name="init">
      <mkdir dir="${dist}"/>
      <mkdir dir="${dist}/WEB-INF"/>
      <mkdir dir="${dist}/WEB-INF/classes"/>
      <copy todir="${dist}" >
         <fileset dir="${src}">
            <include name="*.jsp" />
            <exclude name="build.xml" />
         </fileset>
```

```
        </copy>
        <copy todir="${dist}/WEB-INF" >
            <fileset dir="${src}">
                <include name="web.xml" />
                <exclude name="build.xml" />
            </fileset>
        </copy>
    </target>

<!--     名为 compile 的 target，它编译 JavaBeans 文件      -->
    <target name="compile" depends="init"
            description="compile the source " >
        <javac srcdir="${src}" destdir="${dist}/WEB-INF/classes"/>
    </target>

<!--     名为 dist 的 target，它将 Web 应用打包      -->
    <target name="dist" depends="compile"
            description="generate the distribution" >
      <jar destfile="${basedir}/FirstAnt.war" basedir="${dist}"/>
    </target>

      <!-- 删除编译和发布时使用的临时目录 -->
    <target name="clean"
            description="clean up" >
      <delete dir="${dist}"/>
    </target>
</project>
```

在这个文件中首先定义了 project 的名称是 SimpleProject 及其属性（默认执行的 target 是 dist、基路径是当前路径），之后定义了两个全局的属性，src 表示源程序目录，另外一个为发布需要的临时目录。

之后定义了 4 个 target，分别完成初始化、编译、发布和清除的工作，由于默认的 target 是 dist，所以，如果在运行时不指定 target 就会运行 dist，dist 依赖于 compile，而 compile 又依赖于 init，所以，先执行 init，然后执行 compile，最后才执行 dist。

25.3　用 Ant 发布复杂 Web 应用

本节继续介绍一个实际使用 Ant 的例子，在这个例子中，需要使用特定的包文件才能编译一些文件，相对复杂一点，本例使用的是第 20 章中使用的 Struts 用户登录的例子。

使用 Ant 之前，先来看一下 Struts 用户登录应用的程序结构，如图 25.5 所示。

25.3.1　build.xml 文件

build.xml 的作用是创建一个临时目录，然后在这个临时目录下创建 Web 应用，把 Web 应用打包后发布到 Web 服务器。下面是 build.xml 文件的源代码：

```
    <copy todir="${dist.dir}" >
        <fileset dir="${basedir}"     >
            <include name="pages/*.jsp" />
            <include name="*.jsp" />
            <include name="images/**" />
        </fileset>
     </copy>
    <copy todir="${dist.dir}/WEB-INF/classes" >
        <fileset dir="${basedir}">
            <include name="resources/*.properties" />
        </fileset>
     </copy>
    <copy todir="${dist.dir}/WEB-INF/" >
        <fileset dir="${basedir}"     >
            <include name="*.tld" />
            <include name="*.xml" />
            <exclude name="build.xml" />
        </fileset>
     </copy>

 </target>

  <! ---编译文件---->
<target name="compile" depends="init">
   <javac srcdir="${src.dir}" destdir="${dist.dir}/WEB-INF/classes">
   <classpath refid="web.classpath"/>
  </javac>
  </target>
  <! --- 复制类库文件---->
  <target name="copyjar" depends="compile">
  <copy todir="${dist.dir}/WEB-INF/lib">
     <fileset dir="${struts.home}">
         <include name="antlr.jar"/>
         <include name="commons-beanutils.jar"/>
         <include name="commons-digester.jar"/>
         <include name="commons-fileupload.jar"/>
         <include name="commons-logging.jar"/>
         <include name="commons-validator.jar"/>
         <include name="jakarta-oro.jar"/>
         <include name="struts.jar"/>
     </fileset>
  </copy>

  </target>
<!--部署 -->
 <target name="deploy" depends="copyjar">
  <jar destfile="${tomcat.home}/webapps/${app.name}.war" basedir="${dist.dir}"/>
 </target>
```

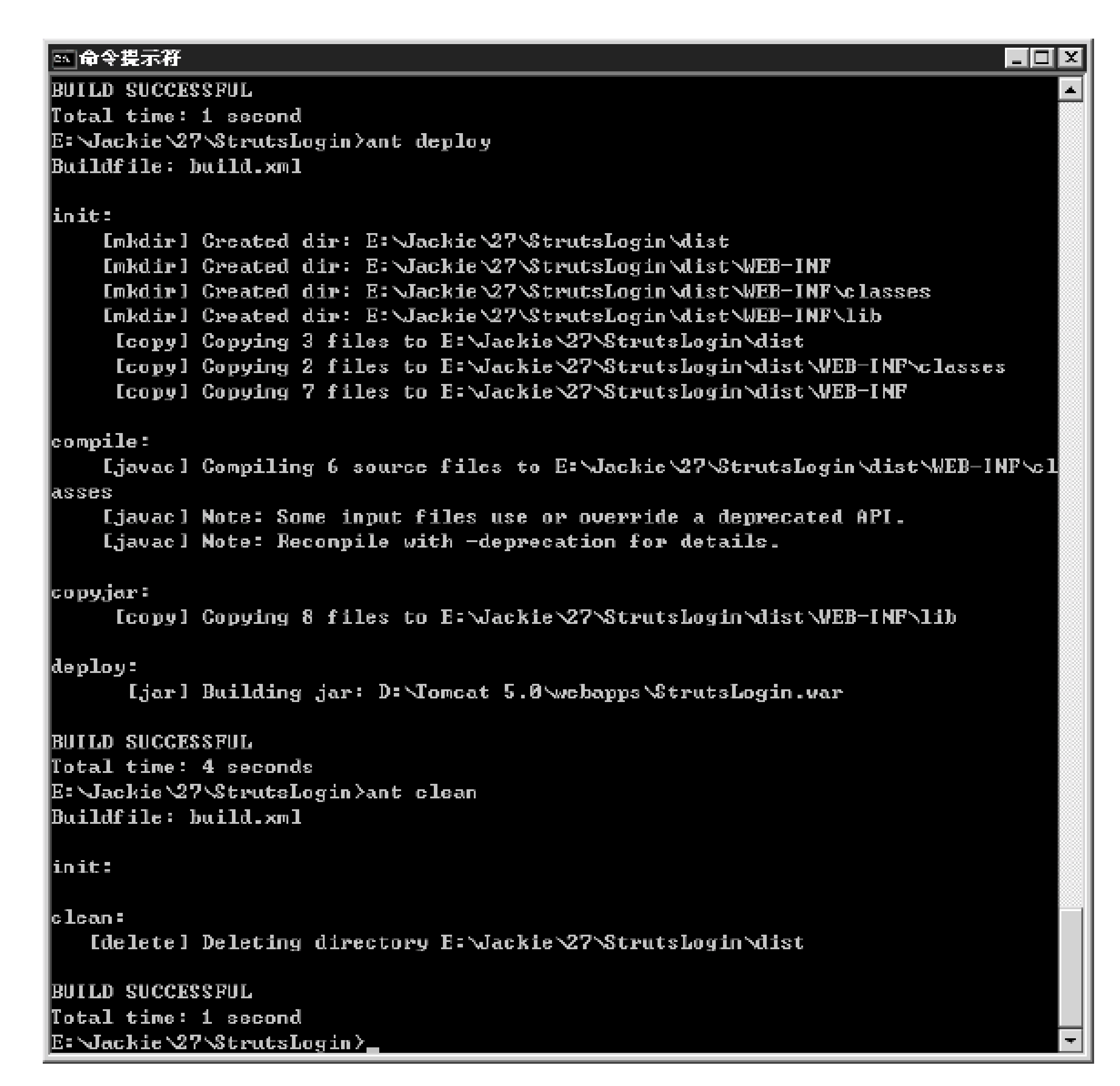


图 25.6　发布 Web 应用

注意： 读者在练习发布这个应用时要注意修改 build.properties 文件中 Tomcat 的主目录和 Struts 的包文件所在的共同主目录。

25.4　小　　结

Ant 工具是 Apache 的一个开放源代码的项目，它是一个非常优秀的软件构建工具。用 Ant 编译或运行比较大的工程是非常方便的，每个工程都有一个包含与这个工程以及需要 Ant 执行的任务信息的文件，这个文件在 Ant 中默认是 build.xml，也可以在 Ant 执行时指定使用的构建文件。在本章中介绍了如何编写 Ant 的构建文件，这个文件的编写是使用 Ant 的重点和难点，希望读者多尝试，体会其用法。在 25.3 节中介绍了使用 Ant 发布 JMail Web 应用的例子，并解释了每个目标完成的工作。

显示它的欢迎界面，如图 26.1 所示。

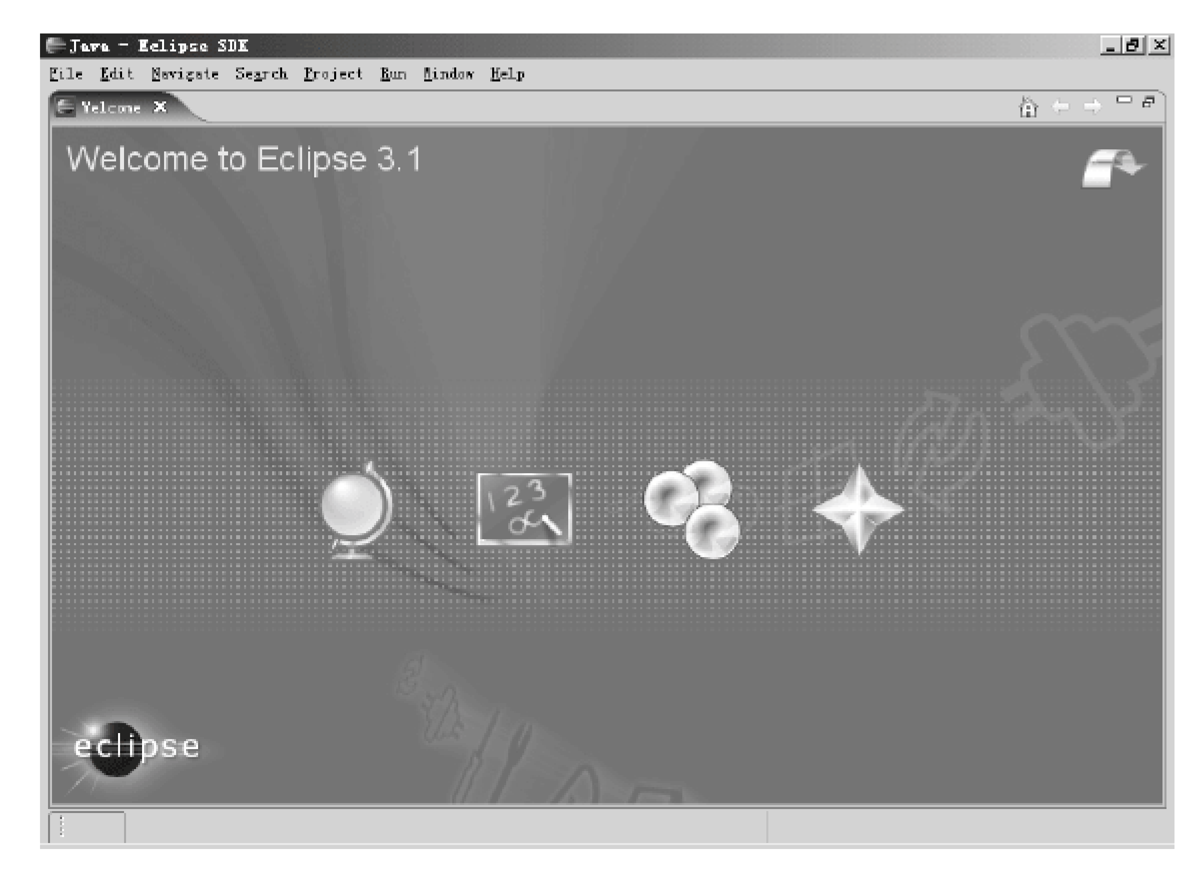


图 26.1　Eclipse 的欢迎界面

关闭欢迎界面后，可以看到 Eclipse 的默认界面（资源透视图），如图 26.2 所示。

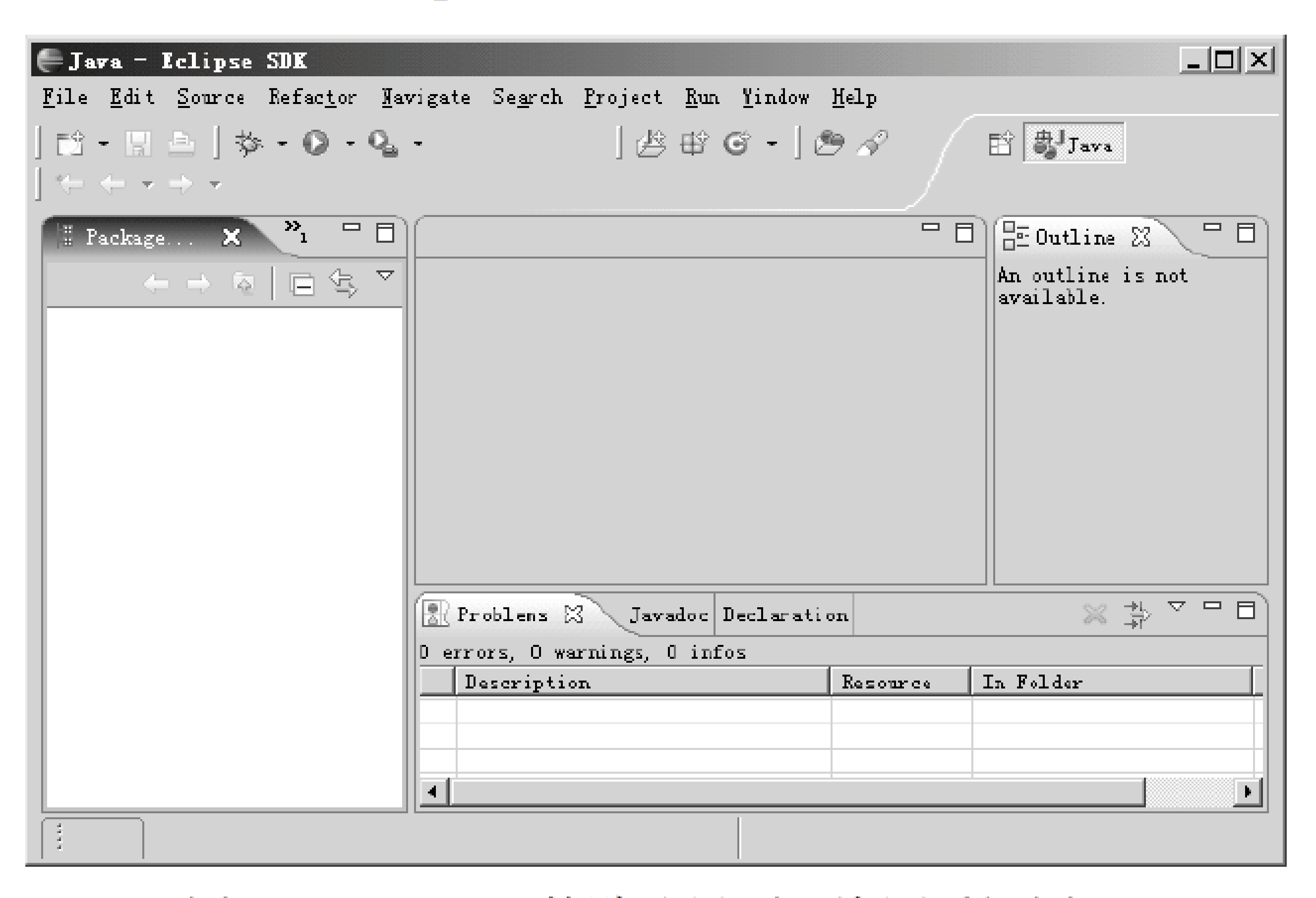


图 26.2　Eclipse 的默认界面（资源透视图）

2．安装多国语言包插件

Eclipse 的一个很大特点就是支持插件，而它的国际化也是通过插件来实现的，只要下载与 SDK 相应的多国语言包插件就可以实现软件的本地化，多国语言包插件的安装和 Eclipse 的安装一样简单，可以按照如下步骤进行：

（1）首先下载多国语言包（下载地址：http://www.eclipse.org），并把多国语言包插件压缩包解压到本地硬盘。

（2）然后把解压目录下 features 和 plugins 文件夹中的所有文件夹复制到对应的 Eclipse 目录中的 features 和 plugins 文件夹下。

（3）重新启动 Eclipse，可以看到 Eclipse 的界面变成了熟悉的中文环境，如图 26.3 所示。

注意： 虽然 Eclipse 支持多国语言，但考虑到进一步的学习，在本章后面所有插件的安装和设置都是按照英文版进行的。

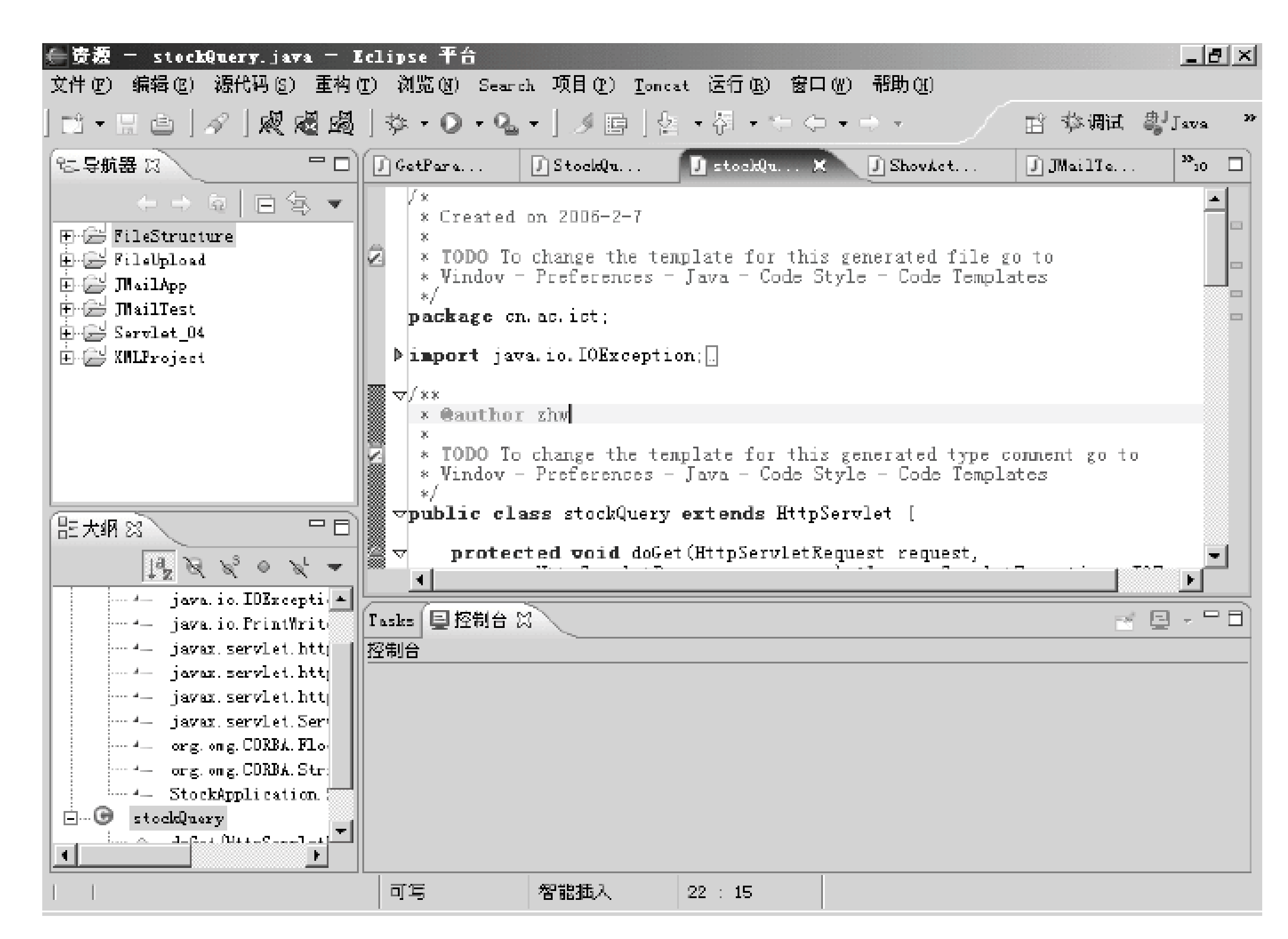


图 26.3　Eclipse 的中文默认界面（资源透视图）

26.2　使用 Eclipse 的 Lomboz 插件开发 JSP

26.2.1　Lomboz 插件介绍

Lomboz 是 Eclipse 的一个主要的开源插件（open-source plug-in），Lomboz 插件能够使 Java 开发者更好地使用 Eclipse 去创建、调试和部署一个 100%基于 J2EE 的 Java 应用服务器。

Lomboz 插件的使用，使得 Eclipse 将多种 J2EE 的元素、Web 应用的开发和最流行的应用服务器结合为一体。

1．Lomboz 的主要功能

Lomboz 的主要功能有如下几项：

- ❑ 使用 HTML、Servlets、JavaServer^{TM} Page（JSP）等方式建立 Web 应用程序。
- ❑ JSP 的编辑带有高亮显示和编码助手。
- ❑ JSP 语法检查。
- ❑ 利用 Wizard 创建 Web 应用和 EJB 应用。
- ❑ 利用 Wizard 创建 EJB 客户端测试程序。
- ❑ 支持部署 J2EE Web 应用档案（EAR）、Web 模块文件（WAR）和 EJB 档案文件（JAR）。
- ❑ 利用 XDoclet 开发符合 EJB 1.1 和 EJB 2.0 的应用。
- ❑ 能够实现端口对端口的本地和远程的测试应用服务。
- ❑ 能够支持所有的有可扩展定义的 Java 应用服务。
- ❑ 能够利用强大的 Java 调试器调试正在运行的服务器端代码（JSP&EJB）。
- ❑ 通过使用 Wizard 和代码生成器提高开发效率。
- ❑ 创建 Web 服务客户端的 WSDL 形式的文件。

的执行环境等信息。

（2）在图 26.5 中左边的树中单击 Server 左边的“＋”，在其子树中选择 Installed Runtime 项，在右面的面板中出现其配置信息，单击 Add 按钮，选择使用的服务器，作者使用 Apache Tomcat v5.0.28，根据提示配置服务器安装主目录和 JSDK 的安装目录等信息，配置完成后，效果如图 26.6 所示。

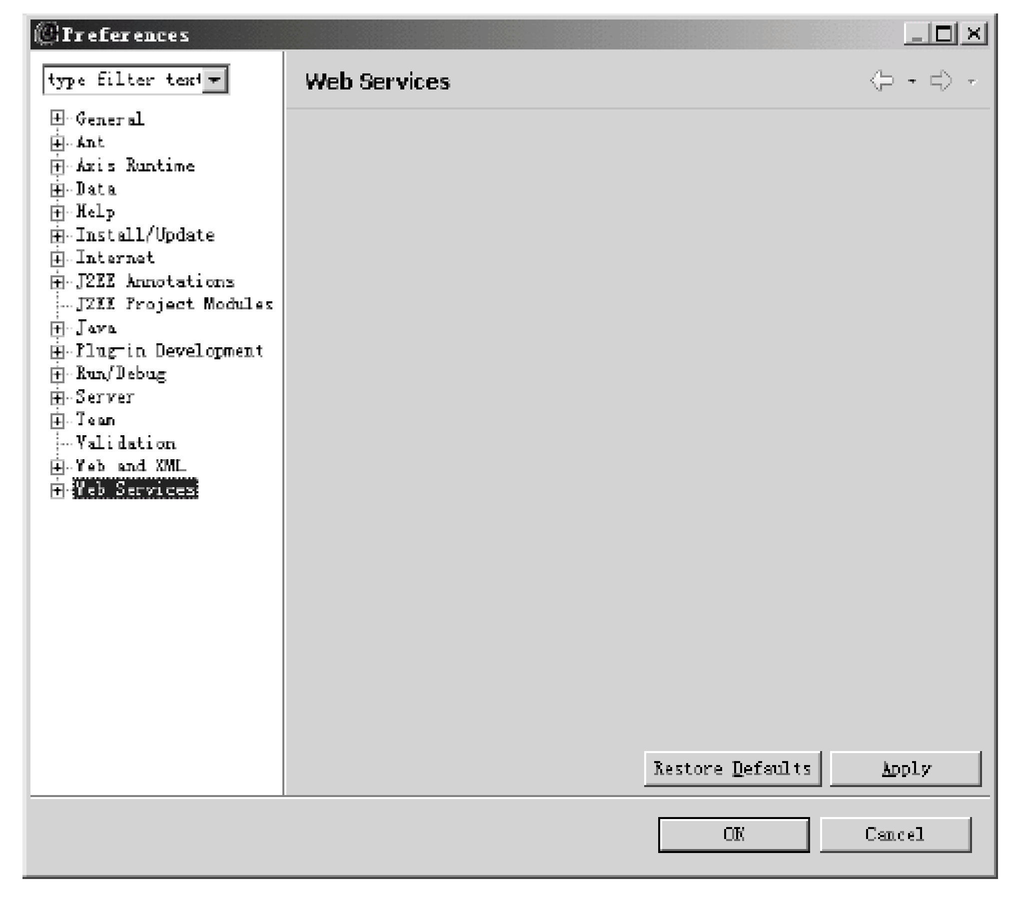
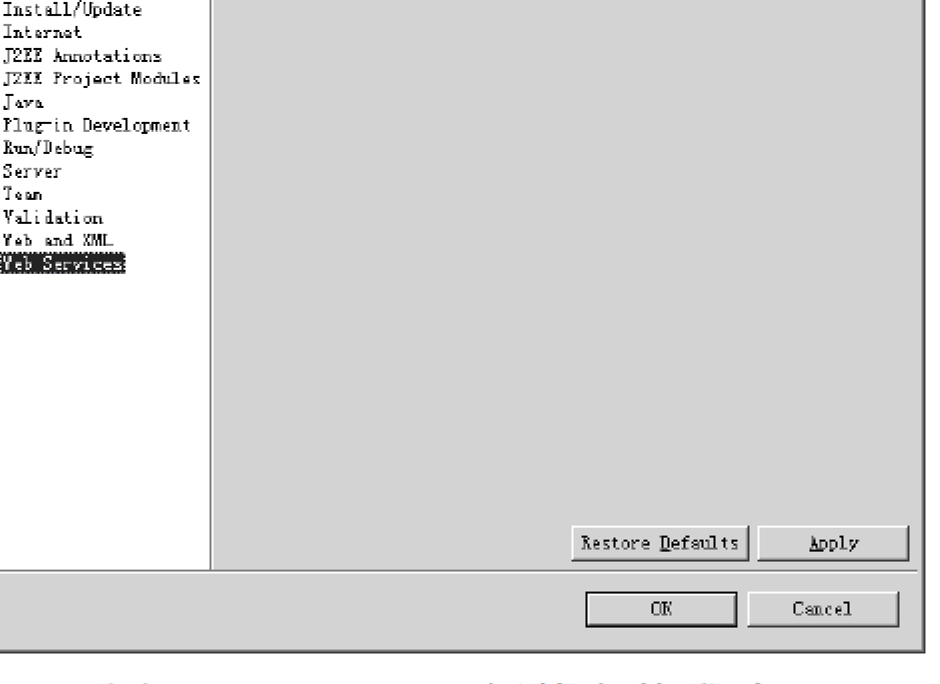



图 26.5　Lomboz 插件安装成功

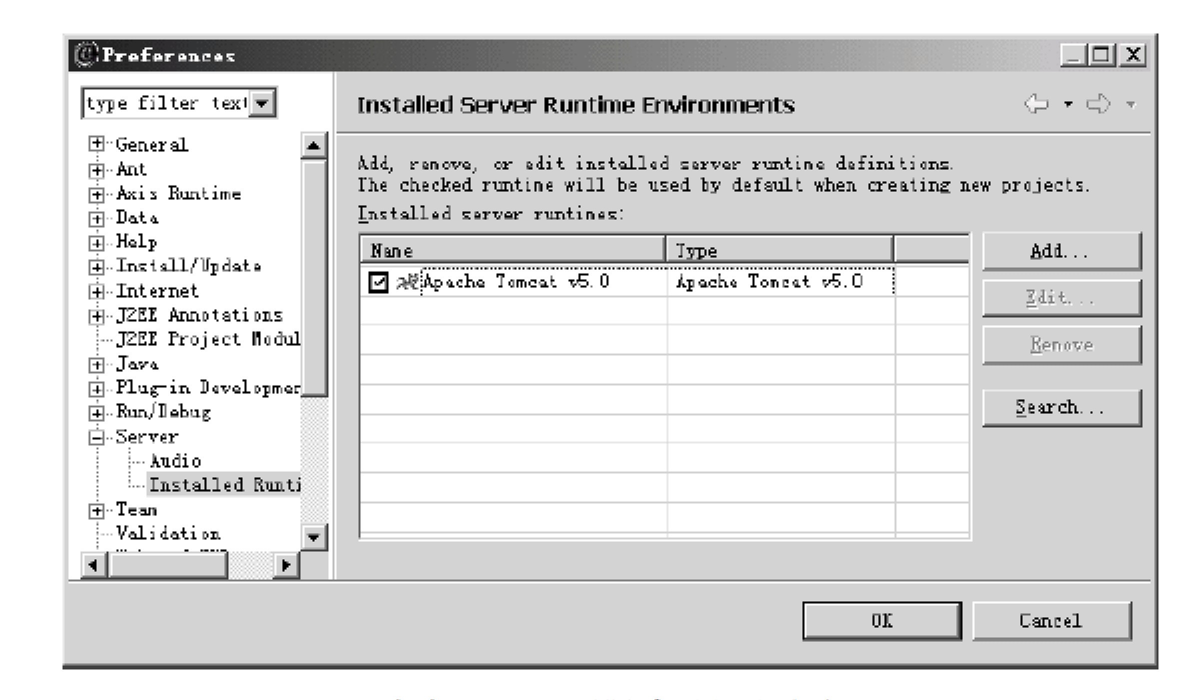


图 26.6　服务器配置

经过上面的配置后，Lomboz 插件需要的配置就完成了。

26.2.3　安装 Tomcat 插件

1．Tomcat 插件的安装

从 http://www.sysdeo.com/上免费下载 Tomcat 插件（目前最高版本为 3.1，适合应用于 Eclipse 3.0 和 Eclipse 3.1 版本），下载后解压缩压缩文件到临时目录，并把临时目录中 com.sysdeo.eclipse.tomcat_X.X.X（X.X.X 为版本号）文件夹复制到 Eclipse 主目录下的 plugins 文件夹就可以了。

注意：如果安装新的插件后，在保证版本正确的情况下，可以把安装目录下 configuration 文件夹中的 org.eclipse.update 目录删掉，然后重新启动 Eclipse，就可以找到新安装的插件了。

2．Tomcat 插件初始化设置

安装成功后，为了能在 Eclipse 中使用 Tomcat 插件，还需要进行一定的初始化设置。

（1）选择工具栏的 Window 菜单命令，在弹出的下拉菜单中选择 Preferences 命令。

（2）在弹出窗口左边的框中选择 Tomcat 项，选择计算机中安装的 Tomcat 版本，并设置 Tomcat 安装的主目录。

（3）单击 Tomcat 项左边的“＋”，单击 Advanced，设置 Tomcat 安装的主目录。

（4）单击 JVM settings 项，在其设置面板中单击 classpath（Before generated classpath）项右边的 Add JAR/ZIP 按钮，添加 JDK 安装目录中 lib 目录下的 tools.jar 文件，单击 Boot

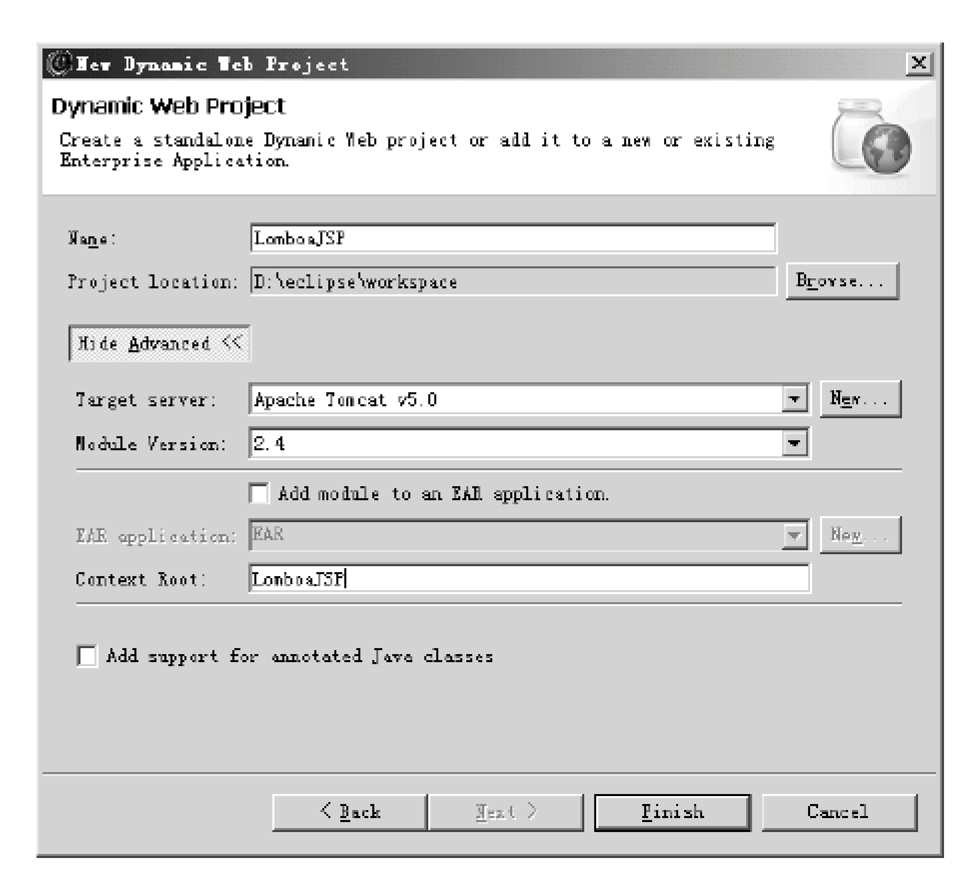


图 26.9　选择新建的服务器

（8）单击 Next 按钮，在配置面板中确保新建的项目成为被配置的项目，也就是出现在右边的栏中。

（9）单击 Finish 按钮，完成设置，这时可以看到 Tomcat 开始启动。

（10）Tomcat 启动完成后，这时会出现一个内嵌的浏览器，并提示“内部服务器错误”，这时可以修改浏览地址为 http://127.0.0.1:8080/LomboaJSP/，这样就可以访问了，因为项目中还没有任何文件，所以只是显示 Directory Listing For，页面显示如图 26.10 所示。

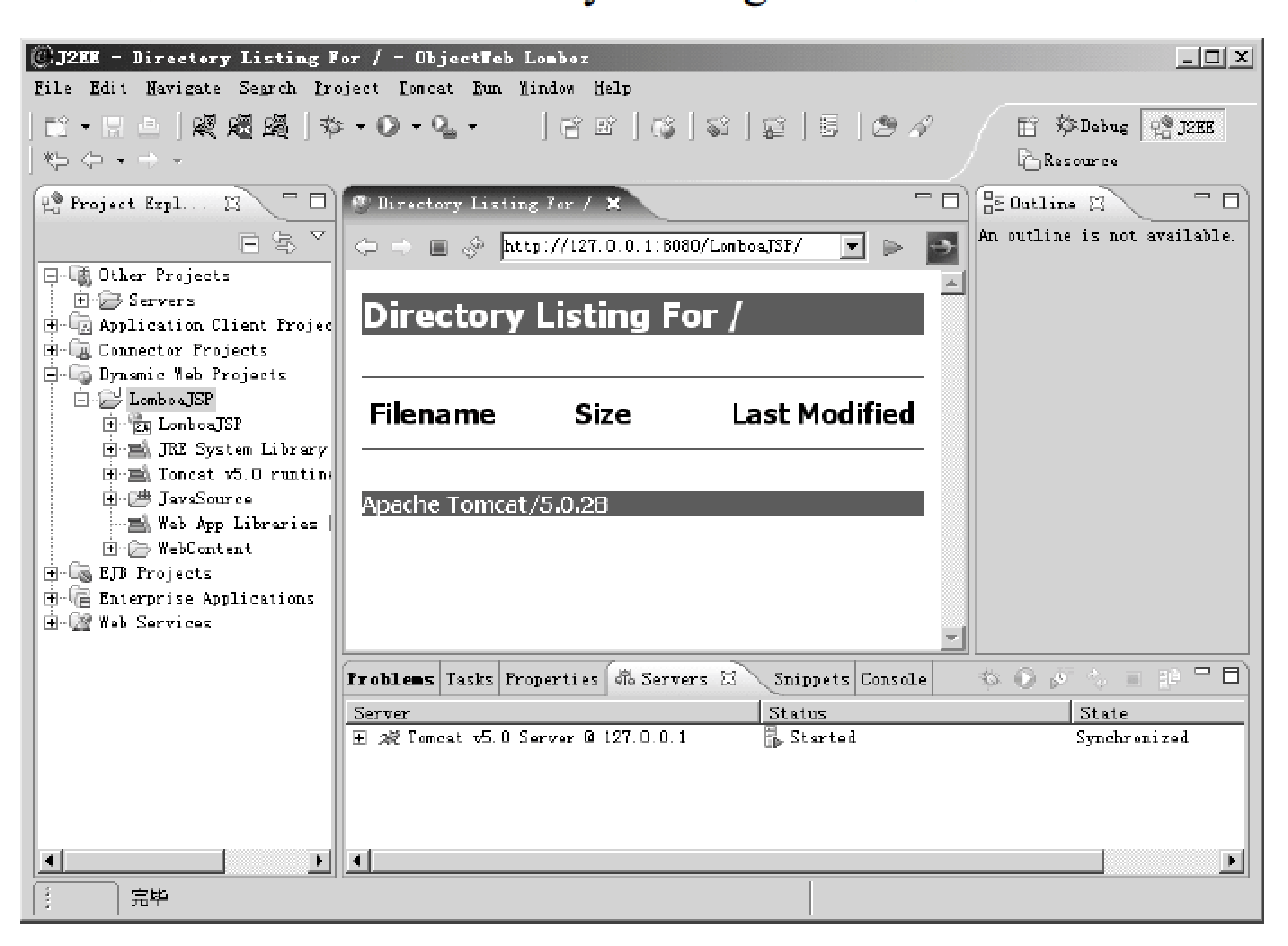


图 26.10　验证项目配置

注意： 服务器 Host 的名称一定要使用 127.0.0.1，不能使用默认的 localhost，否则会出现无法访问资源的错误。

2．建立 JSP 文件

（1）选择工具栏中的 File 命令，再选择 New/Other 命令，在弹出的对话框中选择 JSP。

（2）在弹出的面板中选择 JSP 文件的父文件夹为 LomboaJSP/WebContent，并输入 JSP 文件的文件名为 HelloWorld.jsp，效果如图 26.11 所示。

（3）单击 Next 按钮后，选择使用的 JSP 模板，完成 JSP 文件的建立。

此行作为一个断点，这时这一行最左边会出现一个淡蓝色的点。

（3）选择 Windows/Open Perspective/Debug 命令，进入调试视图模式。

（4）在嵌入的浏览器中输入要调试的 JSP 文件的访问路径，然后查看 JSP 文件，可以看到断点行左边的蓝点变成了向右指的箭头，也就是程序执行到这一行并停止了，效果如图 26.13 所示。

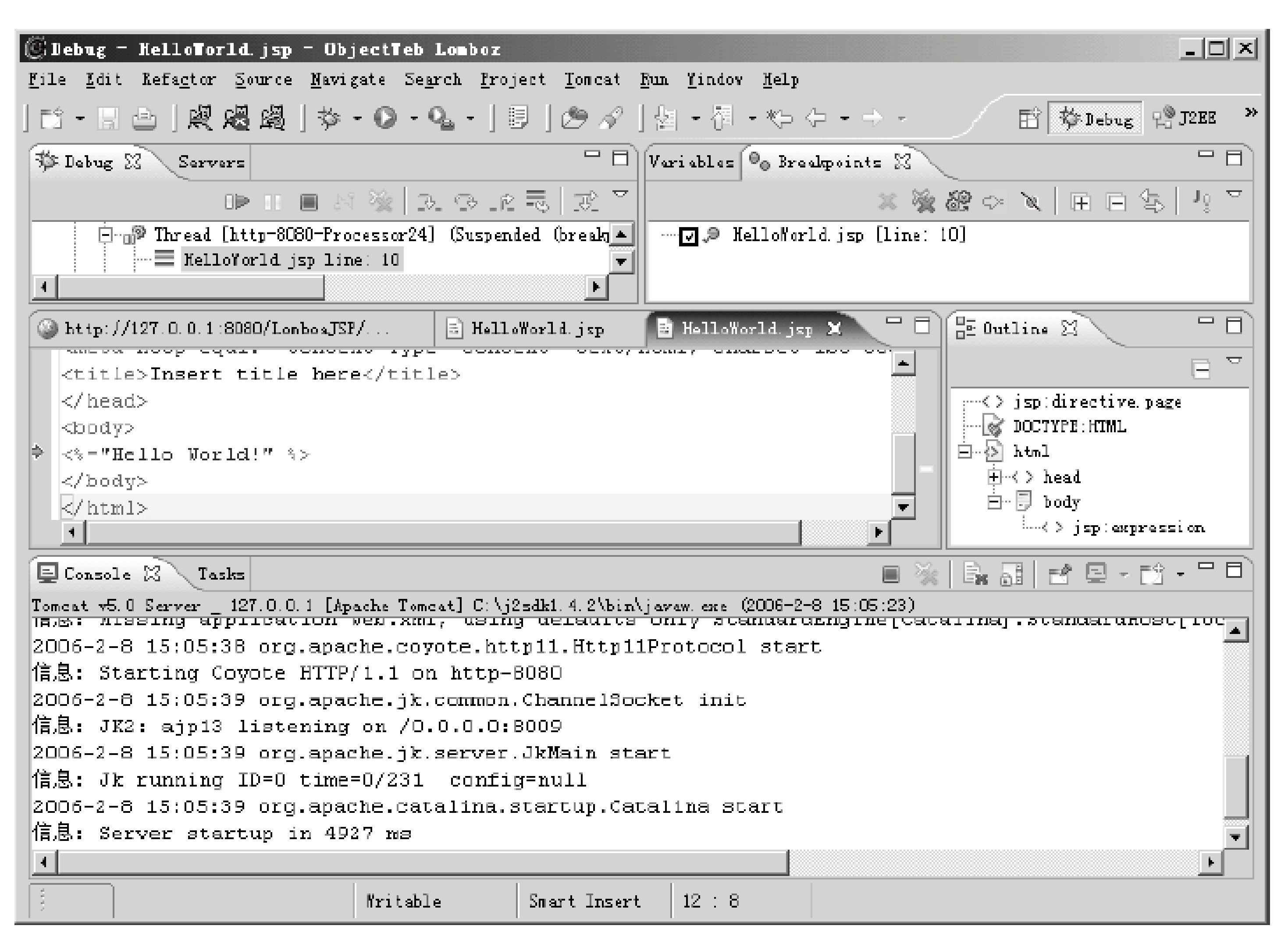


图 26.13　调试 JSP 文件

26.3　小　　结

本章简单介绍了如何使用 Eclipse、Tomcat 和 Lomboz 结合开发 JSP 程序，并使用 Lomboz 调试功能。Tomcat 和 Lomboz 只是 Eclipse 众多插件中的两个，还有很多非常有价值的插件可以使用，也可以在很大程度上方便 JSP 的编程，即使本章中介绍的这两个插件还有很多东西是在开发 JSP 时可以用到的，读者应注意多实践。

- ❑ 用于 J2EE™ 用程序快速开发的 EJB 2.0 的可视化设计器。
- ❑ 面向如下领先应用程序服务器的分发：Borland 企业级服务器、BEA Weblogic、IBM WebSphere 和 iPlanet。
- ❑ 简化数据库应用程序开发和分发的向导、工具和组件。
- ❑ Web 应用程序开发和采用 JSP 和 Servlets 的分发。
- ❑ UML 代码可视化。
- ❑ 重构（refactoring）与单元测试。
- ❑ 集成领先版本的控制系统。
- ❑ 跨设备公布和集成商务数据的 XML 工具。

27.2　使用 JBuilder 开发 JSP（实例）

在本节中介绍如何使用 JBuilder 开发 JSP 程序，以一个用户登录验证为例，JSP 页面负责视图的显示，另一个 Servlet 负责验证，在本节先介绍如何开发 JSP 页面。

27.2.1　建立工程及相关准备工作

要使用 JBuilder 开发一个完整的项目，首先要建立一个工程，用来容纳所开发的程序，读者可以按照如下步骤建立工程。

（1）运行 JBuilder 程序后，在主界面的标题栏中选择 File/New 命令或者在主界面中直接按 Ctrl+N 组合键打开 Object Gallery 面板。

（2）在 Object Gallery 面板的左边一栏中选择 Project 项，在右边出现多个可以选择的工程项目，在本实例中双击第一个 Project 项，建一个新的空普通工程，显示页面如图 27.1 所示。

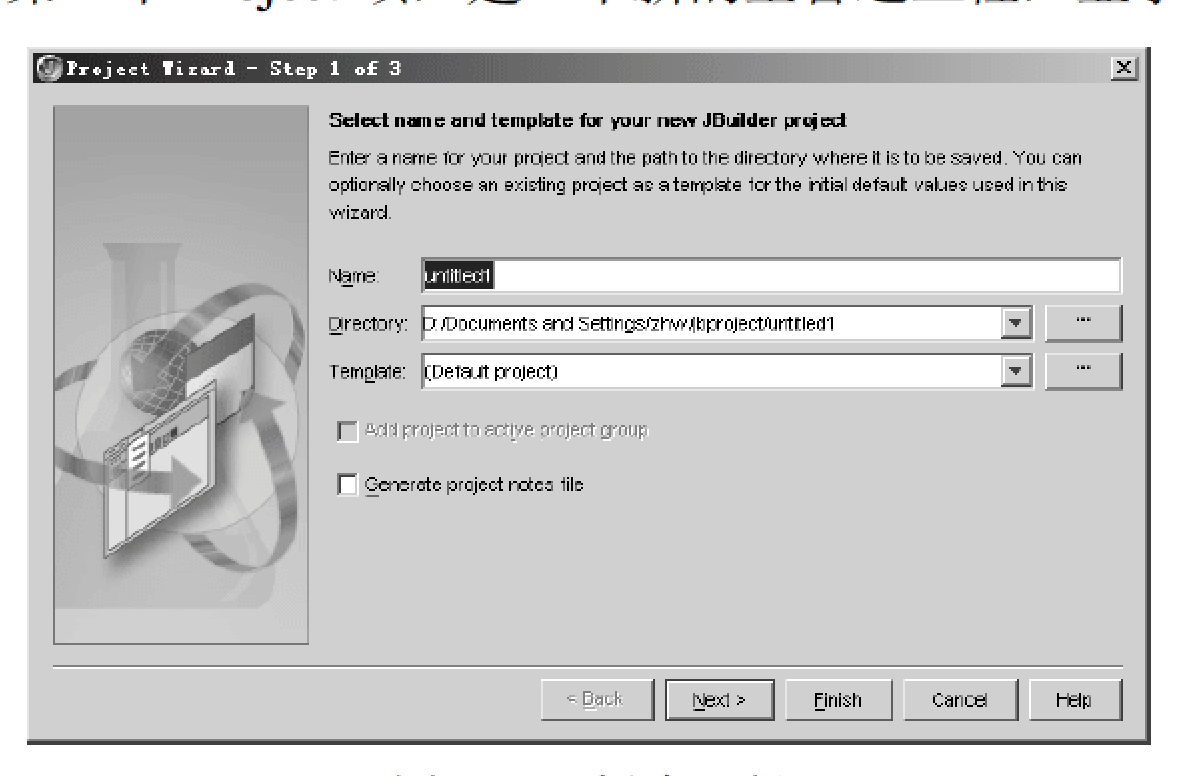


图 27.1　新建工程

（3）修改工程的名称（本例中建议改为 HelloJBuilder）和保存的路径，单击 Next 按钮。

（4）在之后两步出现的面板中根据提示项进行相应的修改，也可以先使用默认设置，在实际开发中根据需要再更改。

（5）这样就新建了一个工程。

27.2.2　设置项目相关属性

在本节中介绍一些 JBuilder 的设置方法，虽然并不一定与要介绍的例子直接相关，但也是一些常用设置。

1．新配置服务器

新配置服务器是开发 Web 应用经常用到的操作，毕竟 JBuilder 不可能包含所有的服务器配置（JBuiler 2005 中只用 Tomcat 4 和 Tomcat 5，其中 Tomcat 5 是 Tomcat 5.0.27），在下面的例子中以新建一个 Tomcat 5.0.28 服务器为例演示如何新建一个新的服务器。读者可以按照如下步骤新建服务器。

注意： 新配置的服务器必须在本机中安装，例如本例中新配置一个 Tomcat 5.0.28 服务器，那么读者的机器上就要安装这个软件。

（1）运行 JBuilder 程序后，在主界面的标题栏中单击 Enterprise 项，选择 Configure Servers，打开 Configure Servers 面板。

（2）在左边的列表中任选一个可用的服务器（显示为黑色，不可用的显示为灰色），单击面板左下角的 Copy…按钮，弹出一个 Copy Server Wizard 对话框，如图 27.2 所示，在各个栏中设置完成后单击 OK 按钮。

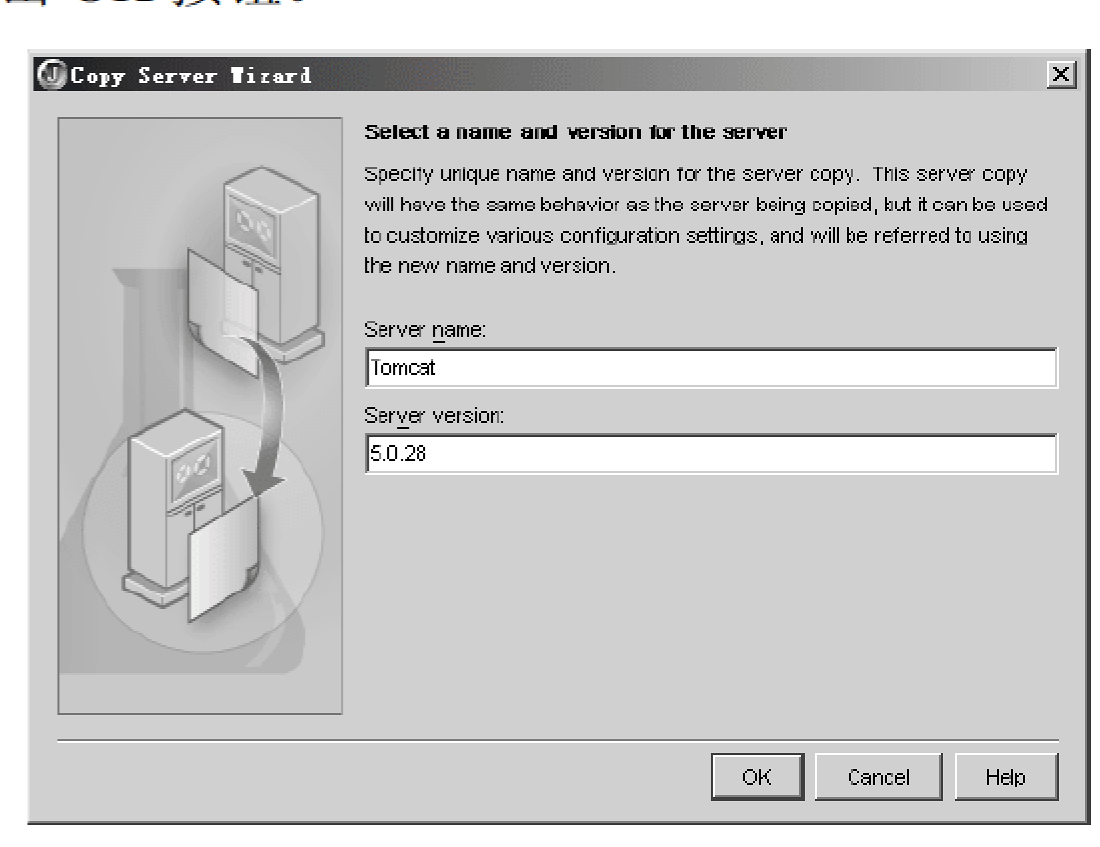


图 27.2　设置新服务器

（3）可以看到在 Configure Servers 面板的左边一栏中出现一项 Tomcat 5.0.28，选中这一项，并在右边的各个栏中设置相关属性。

（4）首先设置服务器的安装目录，笔者的 Tomcat 5.0.28 安装在 D:\Tomcat 5.0；然后是设置启动时使用的类，对于 Tomcat 而言是 org.apache.catalina.startup.Bootstrap，如果读者设置其他服务器可以参考其相关文档；之后还可以指定虚拟机参数和服务器参数；下面还可以设置服务器的源程序、相关文档和需要的库文件等项，这些可以根据需要设置，本例中采用默认设置。

（5）单击 OK 按钮完成设置，这样就可以在新建 Web Module 时选择这个新配置的服务器了。

2．设置项目属性

按照上面的介绍新建完成一个空的项目后，往往还会有很多的属性需要更改，下面对一些设置进行简单介绍。

（1）在 Project 栏中选择需要更改属性的工程，右击这个工程的名字，在弹出的菜单中选择 Properties…命令，出现属性设置面板。

（2）在属性设置面板中有很多属性可以设置，这里不一一介绍，只选择几项介绍。

（3）修改编译选项：在左边栏中选择 Build 下的 Java，可以在右边的面板中设置使用的编译器，调试的选项以及目标虚拟机版本等。

（4）更改代码的格式：左边栏中 Basic Formating 和 Java Formating 项及其子项都是用来设置代码格式的，读者可以更改代码的格式，这样可以保持代码具有统一的风格，便于维护。

（5）修改相关路径：在左边栏中选择 Paths 项，在右边栏中可以修改的路径有源代码的路径、输出的路径、备份的路径以及工作路径等，读者可以根据需要修改，下面将介绍如何添加需要使用的包。

3．给项目添加需要使用的包文件

随着项目的不断变大，可能需要不断添加新的支持包，在本实例中需要使用数据库驱动程序，在下面就以为项目添加数据库驱动程序包为例介绍如何为项目添加需要使用的包文件。

（1）在 Project 栏中选择需要更改属性的工程，右击这个工程的名字，在弹出的菜单中选择 Properties…命令，出现属性设置面板。

（2）在左边栏中选择 Paths 项，在右边页面的下部选择 Required Libraries 选项卡。

（3）单击 Add 按钮，选择 Libraries 选项卡，单击右边的 New 按钮。

（4）弹出 New Library Wizard 对话框，设定其名字为 MySQL Driver，指定包存在的位置为 Project，在后面指定包存在的路径，单击 OK 按钮完成设置。

（5）这样在 Libraries 选项卡下面的框中应该出现一个 MySQL Driver 项就可以了。

27.2.3　新建 Web Module

（1）运行 JBuilder 程序后，在主界面的标题栏中选择 File/New 命令或者在主界面中直接按 Ctrl+N 组合键打开 Object Gallery 面板。

（2）在 Object Gallery 面板的左边一栏中选择 Web 项，在右边出现多个可以选择的工程项目，在本实例中双击第一个 Web Module（WAR），建立一个新的 Web Module，显示页面如图 27.3 所示。

（3）在第一个下拉列表中选择这个 Web Module 使用的服务器，在图 27.3 中可以看到只有两项可以选择，读者可以根据需要自己添加使用服务器（见 27.2.2 节介绍）。

（4）单击 OK 按钮，出现 Web Module Wizard 面板，一共分 3 步，在这些面板中可以修改 Web Module 的名字（本例中笔者设置为 LogInMod）、存放的目录（本例中设置为该

工程所在目录下的 LogInMod 目录）、使用的 Servlet 版本（本例为 2.4 版）和 JSP 版本（本例为 2.0 版本），访问使用的 Context 路径（本例为 LogInMod）等，这里不作详细介绍。

（5）单击 Finish 按钮后，会出现一个该 Web Module 的配置界面，在其中可以设置会话的超时时间、图标等信息，对应于 Web 应用的 web.xml 文件中的某个元素，读者可有选择地进行设置，本例中均使用默认设置。

（6）这样一个 Web Module 就建好了。

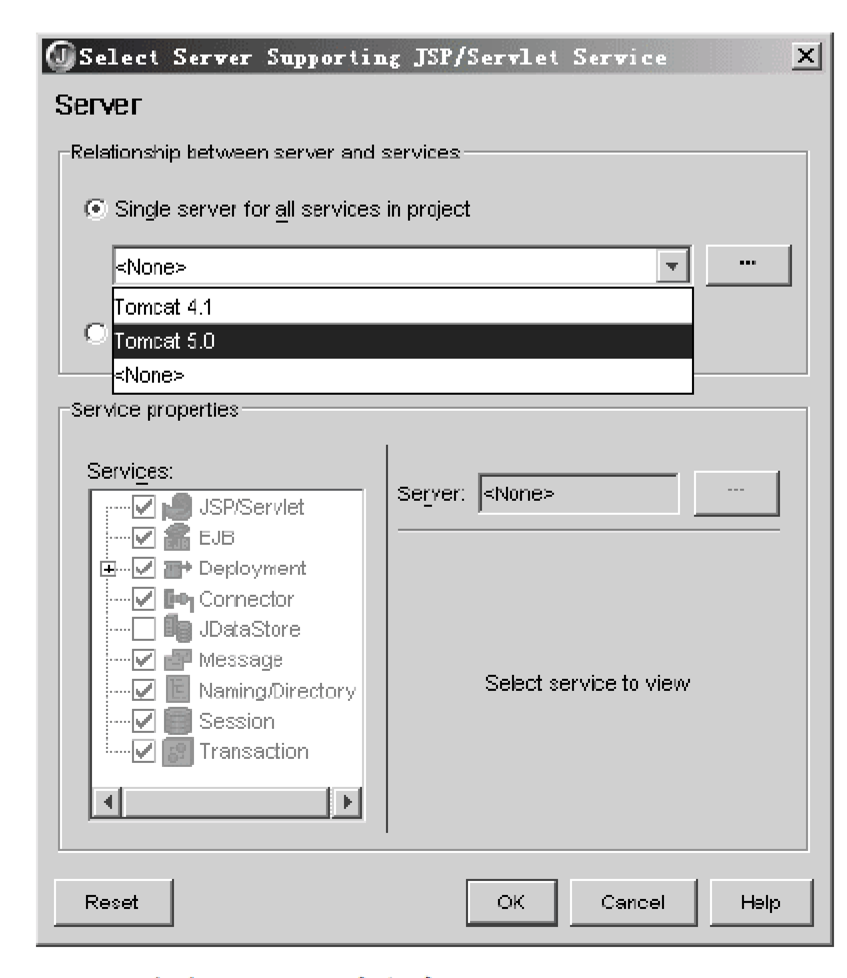


图 27.3　新建 Web Module

27.2.4　开发 JSP

在本小节中介绍如何使用 JBuilder 的 JSP Wizard 创建一个 JSP 页面，读者可以按照如下步骤进行。

（1）运行 JBuilder 程序后，在主界面的标题栏中选择 File/New 命令或者在主界面中直接按 Ctrl+N 组合键打开 Object Gallery 面板。

（2）在 Object Gallery 面板的左边一栏中选择 Web 项，在右边出现多个可以选择的工程项目，在本实例中双击 JSP 项，建立一个新的 JSP 文件，弹出 JSP Wizard 页面。

（3）在第 1 个面板的各个设置中指定 JSP 文件的名字为 login.jsp。

（4）在第 2 个面板中编辑 JSP 文件的一些细节，如背景颜色、是否产生一个 FORM 表单以及使用何种 MVC 框架等，本例中使用默认设置。

（5）在第 3 个面板中可以添加使用的 Bean，本例不使用任何 Bean。

（6）在第 4 个面板中可以设置 JSP 运行环境配置，本例中不需要。

（7）到此，一个 JSP 页面就建好了，读者可以在出现的 JSP 文件编辑区中编辑该 JSP 文件，编辑完成的 JSP 文件（login.jsp）代码如下：

```
<%@ page contentType="text/html; charset=GBK" %>
<html>
<head>
<title>
用户登录
</title>
</head>
<body bgcolor="#ffffff">
  <h1>
          用户登录界面
  </h1>
  <form method="post" action="LogInServlet">
    <br><br>
    用户名：
```

```
      <input type="text" name="username"/><br><br>
      密    码:
      <input type="password" name="password"/><br><br>
      <input type="submit" name="Submit" value="提交">
      <input type="reset" value="重置">
    </form>
</body>
</html>
```

27.3　使用 JBuilder 开发 Servlet（实例）

在 27.2 一节介绍了如何开发 JSP 页面，并介绍了一个简单的用户登录的例子，在本节中开发一个 Servlet 用于对用户信息进行确认。读者可以按照如下步骤进行：

（1）运行 JBuilder 程序后，在主界面的标题栏中选择 File/New 命令或者在主界面中直接按 Ctrl+N 组合键打开 Object Gallery 面板。

（2）在 Object Gallery 面板的左边一栏中选择 Web 项，在右边出现多个可以选择的工程项目，在本实例中双击 Standard Servlet 项，建立一个新的 Servlet 文件，弹出 Select Server 对话框。

（3）在其中选择本工程使用的服务器（在 27.2.3 节中新配置的 Tomcat 5.0.28），单击 OK 按钮，弹出 Servlet Wizard 对话框。

（4）在 Servlet Wizard 对话框（分 5 步完成）中设置类名为 LoginCongfirm，包名为 hellojbuilder，使用的 Web Module 是 27.2.2 节中建立的 LoginMod。

（5）在第 2 和第 3 个对话框中设置实现的方法、内容的类型、获取的请求参数（本例中使用两个，分别是 username 和 password，效果如图 27.4 所示）。

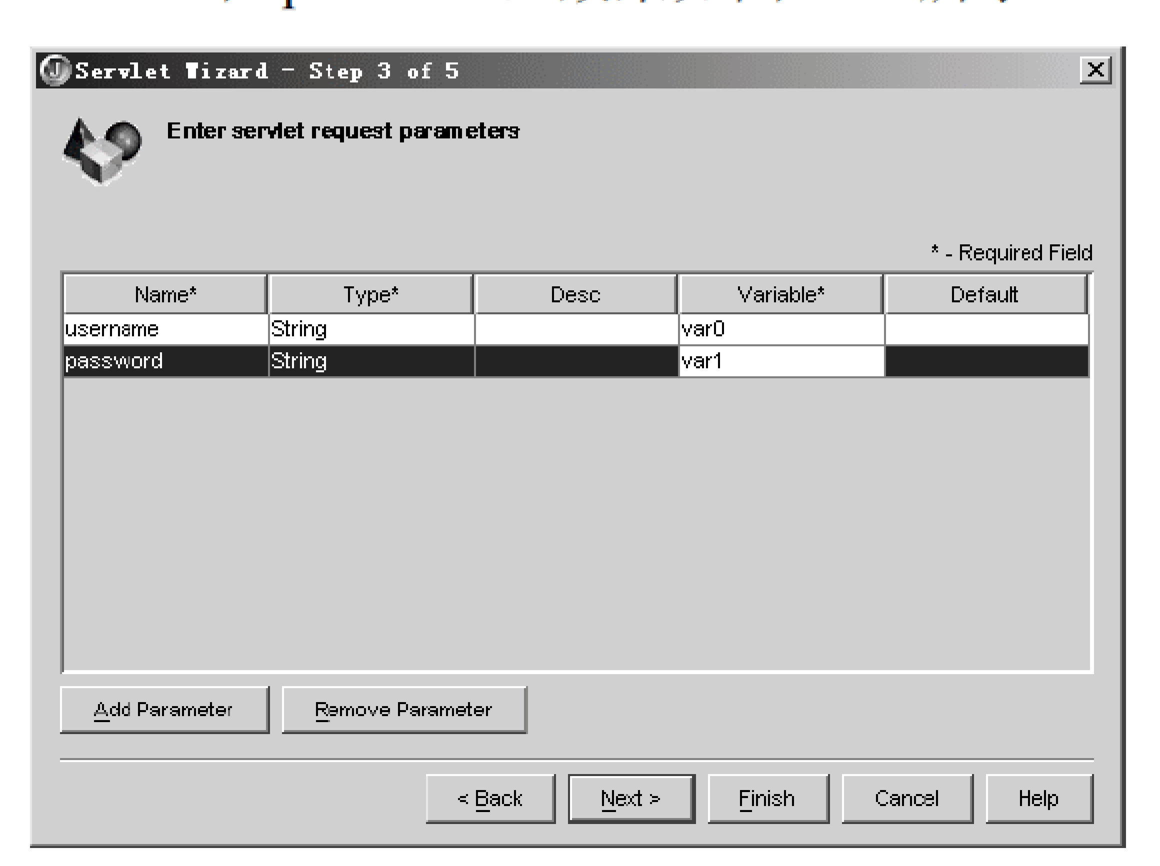


图 27.4　为 Servlet 添加请求参数

（6）在第 4 个页面中指定 Servlet 的名字和映射使用的名字，这里默认使用 LoginServlet 和/LoginServlet，在第 5 个页面中使用默认设置就可以了。

（7）单击 Finish 按钮，完成新建一个 Servlet。

（8）编辑这个 Servlet 的代码，最后编辑完成后，代码如下：

```
package hellojbuilder;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

public class LoginCongfirm extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html; charset=GBK";
    private Connection conn=null;
    private String url = "jdbc:mysql://localhost:3306/userLIB";
    private String user = "root";
    private String passw = "ict";
    private boolean legalUser = false;

    //初始化全局变量，加载数据库驱动程序
    public void init() throws ServletException {
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            conn = DriverManager.getConnection(url,user,passw);
        } catch (ClassNotFoundException ex) {
            this.getServletContext().log("无法找到驱动程序",ex);
        } catch (IllegalAccessException ex) {
            this.getServletContext().log("无法获取访问驱动程序的权限",ex);
            /** @todo Handle this exception */
        } catch (InstantiationException ex) {
            this.getServletContext().log("无法获取连接",ex);
            /** @todo Handle this exception */
        } catch (SQLException ex) {
            this.getServletContext().log("SQL 错误",ex);
            /** @todo Handle this exception */
        }

    }

    //处理 HTTP 的 Get 请求
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        String username = request.getParameter("username");
```

```
    //释放占用的资源
    public void destroy() {
    }
}
```

（9）建立数据库，在 MySQL 数据库系统中建立 userLib 数据库，并建立 userInfo 数据表，使用的 SQL 语句如下：

```
CREATE TABLE 'userinfo' (
  'username' char(20) default NULL,
  'password' char(20) default NULL
)
INSERT INTO `userinfo` VALUES ('zhw','111111');
```

（10）上面的各步都准备好了，下面就可以运行了，选择 Run/Run Project 命令。

（11）这时会出现内嵌的浏览器，在地址栏中输入 http://localhost:8083/LoginMod/login.jsp，并输入用户名（zhw）和密码（111111），页面显示如图 27.5 所示。

注意：输入的用户名和密码只要与数据库中一致即可。

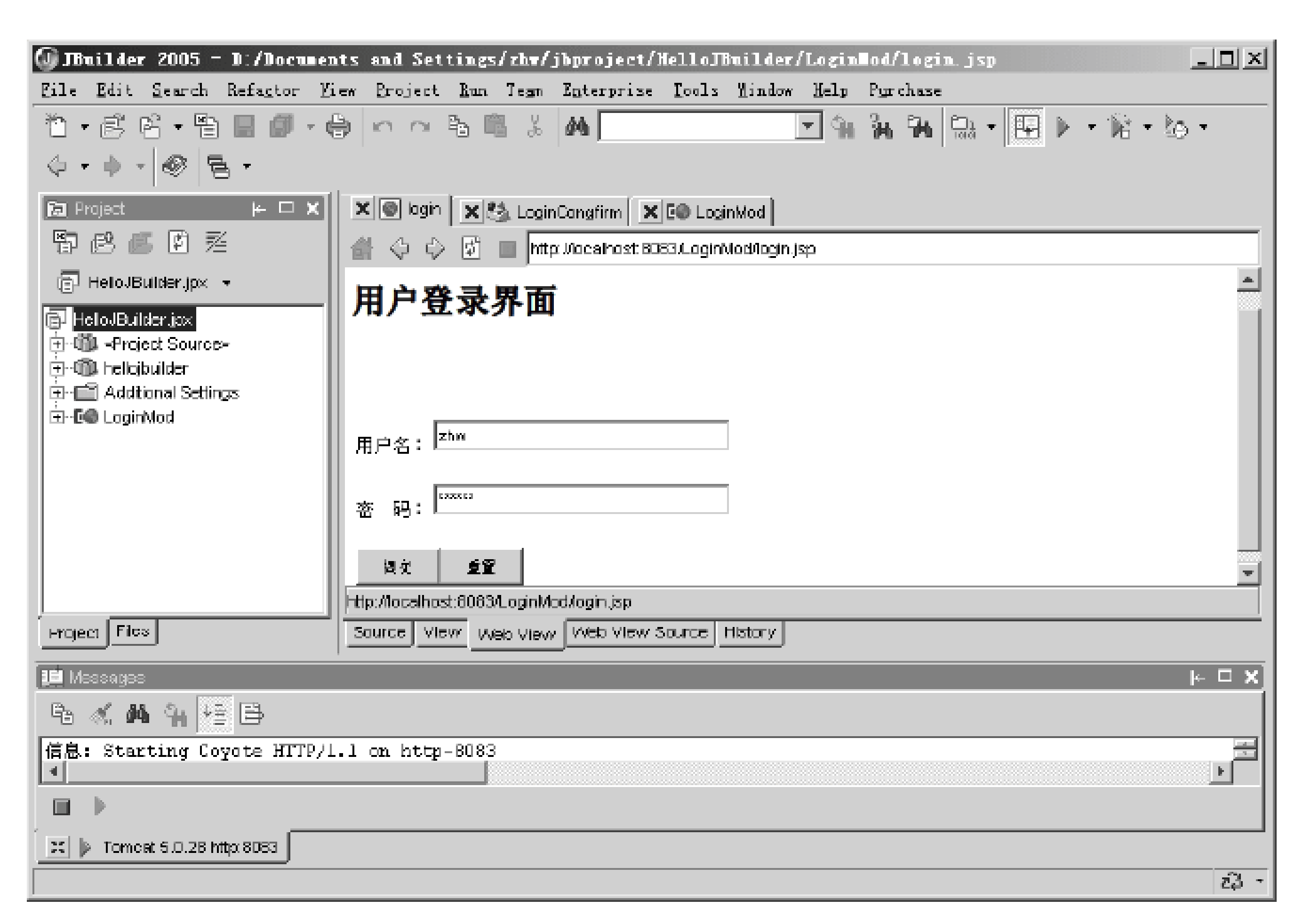


图 27.5　输入用户名和密码

（12）提交后，页面显示如图 27.6 所示。

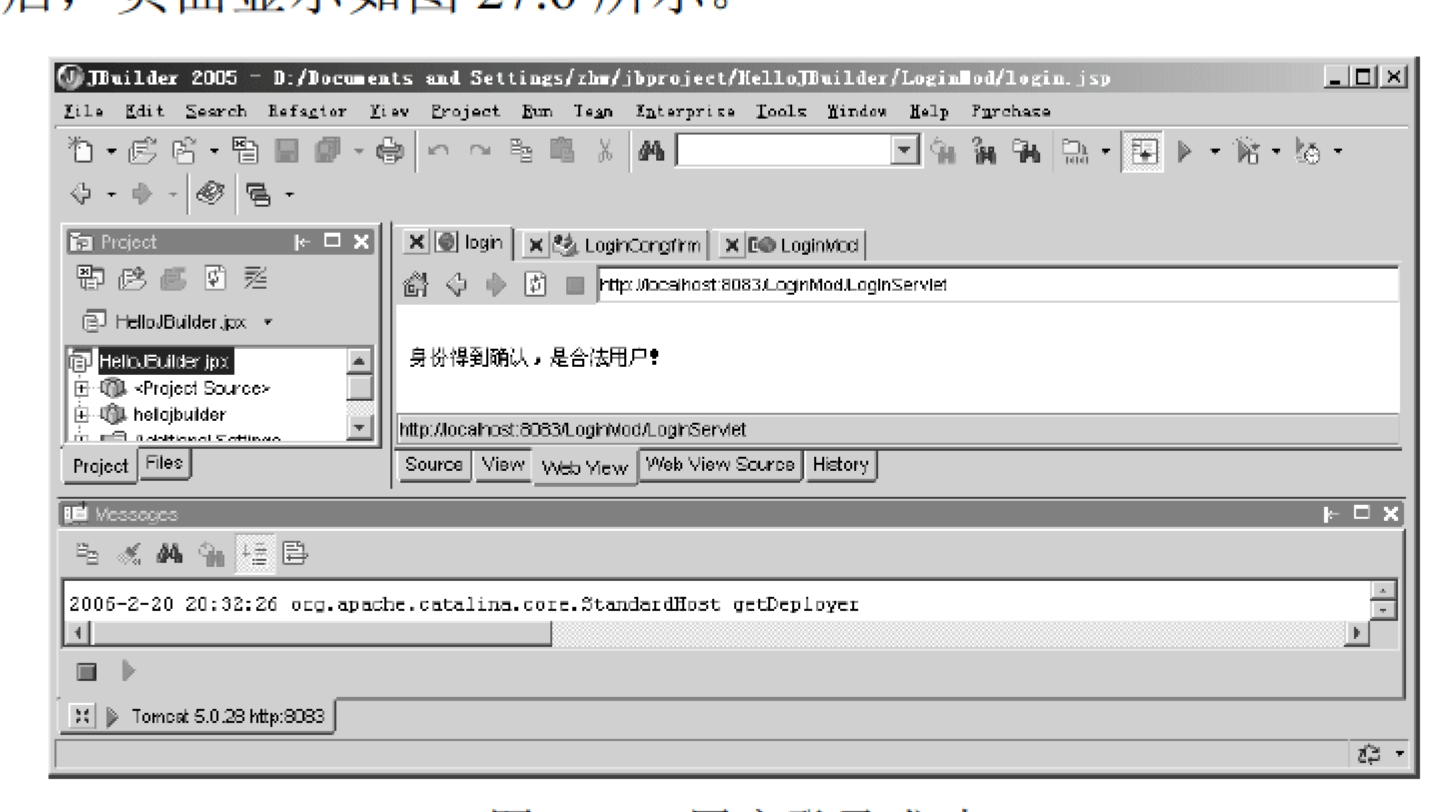


图 27.6　用户登录成功

（6）从图 27.8 中可以看到，查询结果中没有任何记录，而把程序使用的 SQL 语句部分选中，查看查询数据库时使用的 SQL 语句，效果如图 27.9 所示。

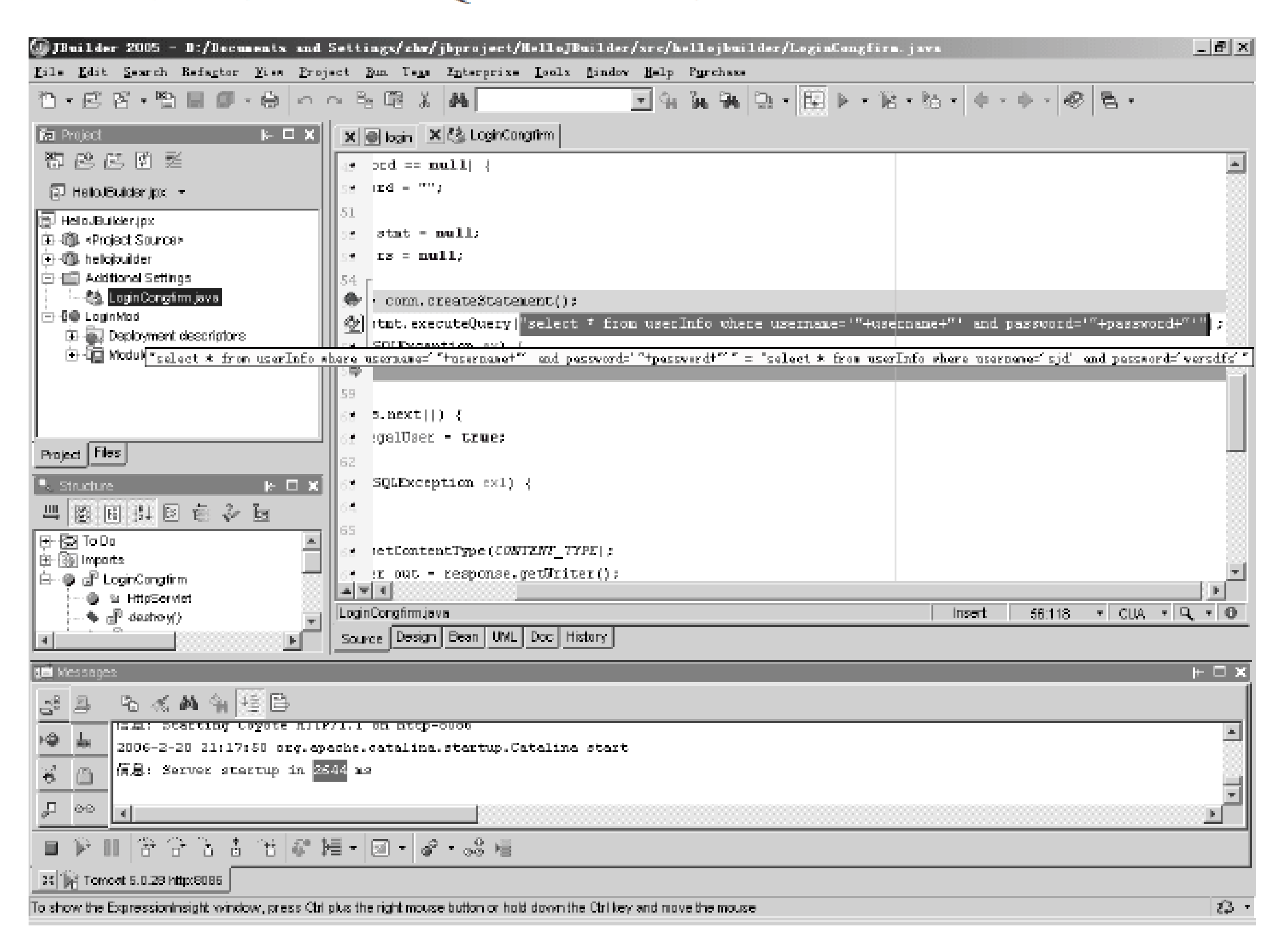


图 27.9　查看 SQL 语句

（7）把使用 SQL 语句记录下来，放到 MySQL 数据库系统上查询，可以看到的确是找不到记录的，也就说明用户的输入错误。

（8）至此一个调试过程就结束了，当遇到其他问题时读者可以采用类似的方法解决。

27.5　小　　结

在本章中介绍了如何使用 JBuilder 这个强大的集成开发环境开发 JSP 的 Web 应用，JBuilder 的功能非常丰富，开发 Web 应用只是其功能的一小部分，读者如果遇到问题可以参考其相关文档，读者应该多使用它开发一些小程序，会发现很多操作都是类似的。

用。使得开发和维护都变得复杂，而上述 J2EE 的三层应用体系结构就解决了这个问题，使得负载得到均衡、性能得到优化。

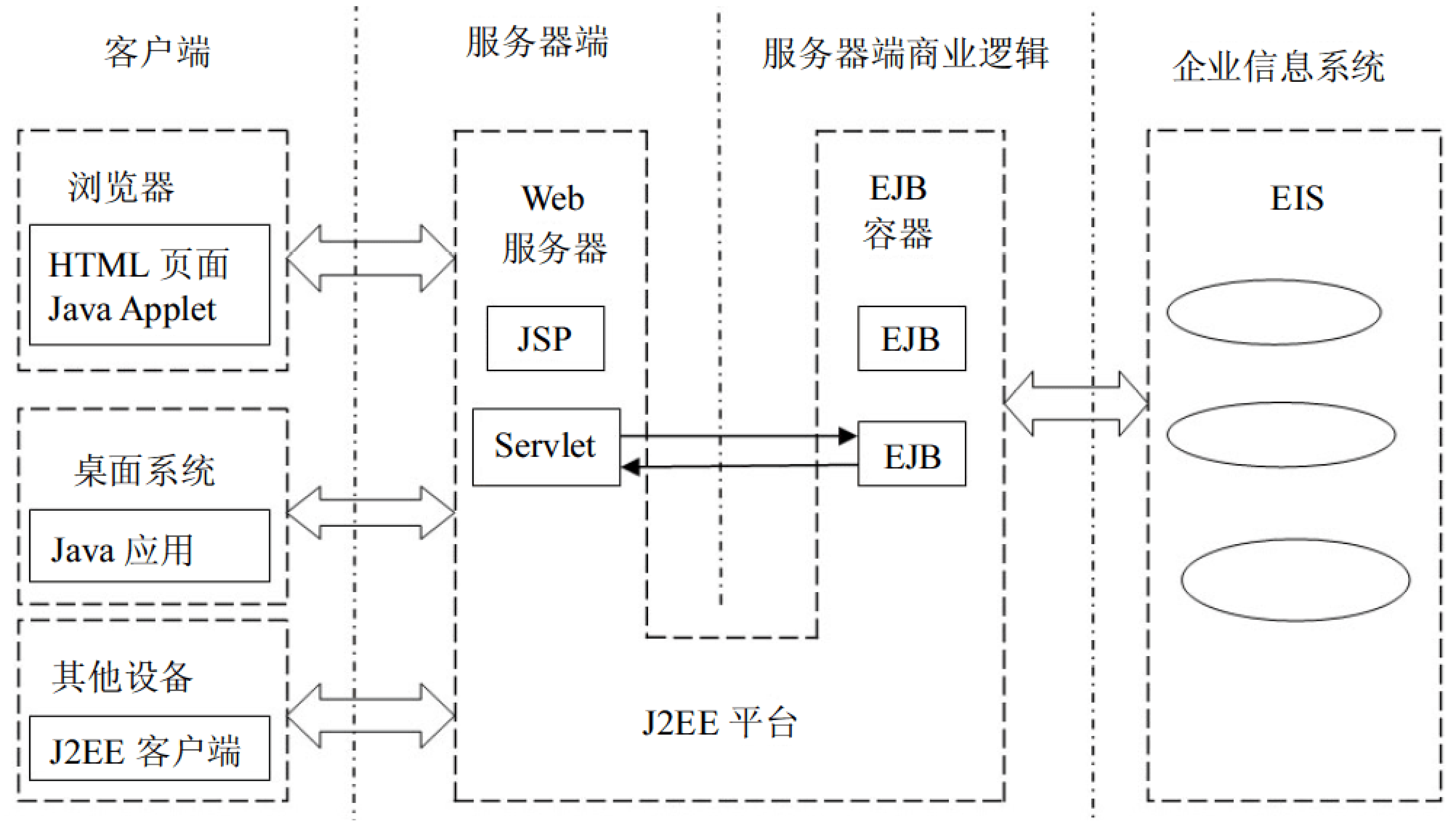


图 28.1　多层 J2EE 应用体系结构

28.1.2　JBoss 入门

通常说的 J2EE 服务器简单来说就是能够提供 JSP 和 EJB 服务器的软件，现在已经有很多的 J2EE 服务器可供选择了，例如 IBM 的 WebSphere 产品，BEA 的 BEA WebLogic 系列以及开源的 JBoss 等，但前两种都是很庞大的软件平台而且收费也很高，在本章中为了演示如何使用 JSP 和 EJB 构建 J2EE 服务，将会使用 JBoss 作为 J2EE 服务器。

1．JBoss 简介

前面介绍过，JBoss 是一个开源的 J2EE 服务器，它的最新版保持并遵循最新的 J2EE 规范。从 JBoss 项目开始至今，它已经从一个 EJB 容器发展成为一个基于 J2EE 的 Web 操作系统，体现了 J2EE 规范中最新的技术。

在 J2EE 应用服务器领域，JBoss 是发展最为迅速的应用服务器。由于 JBoss 遵循商业友好的 LGPL 授权分发，并且由开源社区开发，这使得 JBoss 广为流行。而且，JBoss 应用服务器还具有许多优秀的特质。

- ❑ 它将具有革命性的 JMX 微内核服务作为其总线结构。
- ❑ 它本身就是面向服务的架构（Service-Oriented Architecture，SOA）。
- ❑ 它还具有统一的类装载器，从而能够实现应用的“热”部署和“热”卸载能力。

注意： “热”部署的意思就是部署 Bean 时只需要简单复制 Bean 的 JAR 文件到部署路径下就可以了，如果 Bean 已经被 LOAD，JBOSS 卸载它，然后 LOAD 一个新版本 Bean。

因此，它是高度模块化的和松耦合的。JBoss 用户反馈显示，JBoss 应用服务器是健壮

```
        }
//下面的方法是会话 EJB 的生命周期方法，可以不实现
        public void ejbActivate() throws EJBException, RemoteException {
        }

        public void ejbCreate() throws CreateException{

        }
        public void ejbPassivate() throws EJBException, RemoteException {
        }

        public void ejbRemove() throws EJBException, RemoteException {

        }

        public void setSessionContext(SessionContext arg0) throws EJBException, RemoteException
{

        }
        //具体实现 Remote 接口文件中定义的方法，它返回一个字符串
        public String getWelcome(){
                return "Hello EJB";
        }
}
```

在上面的代码中可以看到 HelloEJBBean 中定义了方法：ejbCreate()、ejbRemove() 、setSessionContext()、ejbPassivate()、ejbActivate()和 ejbPassivate()，这些方法是会话 Bean 中用于维护生命周期的方法，具体介绍见本章后面对会话 Bean 的介绍。

4．编写 EJB 组件发布描述文件

EJB 组件由相关的类文件和 EJB 的发布描述文件构成，ejb-jar.xml 文件是 EJB 组件的发布描述文件。在该文件中定义了 EJB 组件的类型，指明其 Remote 接口、Home 接口和 Enterprise Bean 类对应的 Java 类文件的完整类名。下面是 HelloEJB 这个 EJB 组件的 ejb-jar.xml 文件内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<!--- 定义一个包含 EJB 的 JAR 包 --->
<ejb-jar>
                <description>
                This is Hello EJB example
                </description>
                <display-name>HelloBean</display-name>
<!--- 下面定义 EJB 的信息 --->
                <enterprise-beans>
                        <session>
                                <display-name>Hello EJB</display-name>
                                <ejb-name>HelloEJB</ejb-name>
```

此时，在该目录下会生成 HelloEJB.jar 文件。单独发布此 EJB 组件时把这个 JAR 文件复制到<JBOSS_HOME>/server/default/deploy 下就可以了，如果此 EJB 组件的相关配置都正确，JBoss 启动时会提示正确，否则需要根据 JBoss 提供的错误信息对类文件或配置文件进行修改。

28.1.5　在 Web 应用中访问 EJB 组件

1．编写访问 EJB 组件的 JSP 文件

在本实例中使用 JSP 文件获得 EJB 组件的引用，并调用其可用方法并输出调用的结果：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import="cn.ac.ict.EJBPackage.*"%>
<%@ page import="javax.naming.*"%>

<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>Lomboz JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
try{
//得到初始化上下文
			InitialContext ctx=new InitialContext();
//查找到 EJB 的引用
			Object objRef = ctx.lookup("java:comp/env/ejb/HelloEJB");
//主接口
			HelloEJBHome
home=(HelloEJBHome)javax.rmi.PortableRemoteObject.narrow(objRef,cn.ac.ict.EJBPackage.HelloEJBHome.class);
//创建一个 EJB 的实现类对象
			HelloEJB bean = home.create();

			out.print(bean.getWelcome());
			}catch(Exception ex){
				out.println(ex);
			}
%>
</body>
</html>
```

从上面的代码中，可以看到如下这段代码通过查找 JNDI 名获得该 JNDI 资源的远程引用：

```
InitialContext ic = new InitialContext();
Object objRef = ic.lookup("java:comp/env/ejb/HelloEJB");
```

然后把这个对象转化为 HelloEJBHome 类型：

```
HelloEJBHome
home=(HelloEJBHome)javax.rmi.PortableRemoteObject.narrow(objRef,cn.ac.ict.EJBPackage.Hell
oEJBHome.class);
```

这样就可以调用 home 的 create 方法了，此时，EJB 容器会创建 HelloEJBBean 的实例，并调用其 ejbCreate 方法，然后返回 HelloEJBBean 的远程应用。

```
shopdb = home.create();
```

得到 HelloEJBBean 的远程应用后，就可以调用可用的方法了，在程序中得到欢迎消息并输出到 JSP 页面：

```
                    out.print(bean.getWelcome());
```

2．编写 Web 应用访问 EJB 组件需要的配置文件

要使 Web 应用访问到 EJB 组件需要两个配置文件：web.xml（Tomcat 中 Web 应用共有的文件）和 jboss-web.xml（Web 应用配置在 JBoss 上必需的文件）。

在这个 Web 应用中需要访问 28.1.4 节中开发的 EJB 组件，因此需要在 Tomcat 的 web.xml 文件中提供关于 EJB 组件的信息（使用 ejb-ref 元素），下面是 web.xml 的完整代码：

```
<?xml version="1.0" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

      <!-- ### EJB References (java:comp/env/ejb) -->
      <ejb-ref>
<!-- 访问 EJB 使用的名字 -- >
            <ejb-ref-name>ejb/HelloEJB</ejb-ref-name>
<!-- EJB 的类型 -- >
            <ejb-ref-type>Session</ejb-ref-type>
            <home>cn.ac.ict.EJBPackage.HelloEJBHome</home>
            <remote>cn.ac.ict.EJBPackage.HelloEJB</remote>
      </ejb-ref>

</web-app>
```

在以上代码中声明了对于 EJB 组件 HelloEJB 的应用，ejb-ref-type 声明了所引用的 EJB 组件的类型（Session），home 元素声明了 EJB 的 Home 接口，remote 元素声明了 EJB 的 Remote 接口，在 Web 应用中，可以通过 ejb-ref-name 所指定的名字（不是 JNDI 的名字）来获得 EJB 的引用，代码如下：

```
InitialContext ic = new InitialContext();
Object objRef = ic.lookup("java:comp/env/ejb/HelloEJB");
```

要把 Web 应用发布到 JBoss 服务器上，还需要提供 jboss-web.xml 文件，它指定了

<ejb-ref-name>（在 web.xml 文件中指定的）和<jndi-name>的映射关系。jboss-web.xml 文件内容如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<jboss-web>
    <ejb-ref>
        <ejb-ref-name>ejb/HelloEJB</ejb-ref-name>
        <jndi-name>ejb/HelloEJB</jndi-name>
    </ejb-ref>
</jboss-web>
```

在这个文件中对访问 EJB 组件的名字和 JNDI 名字进行了映射，启动这个 JNDI 名必须在上节中介绍的 jboss.xml 文件中定义。

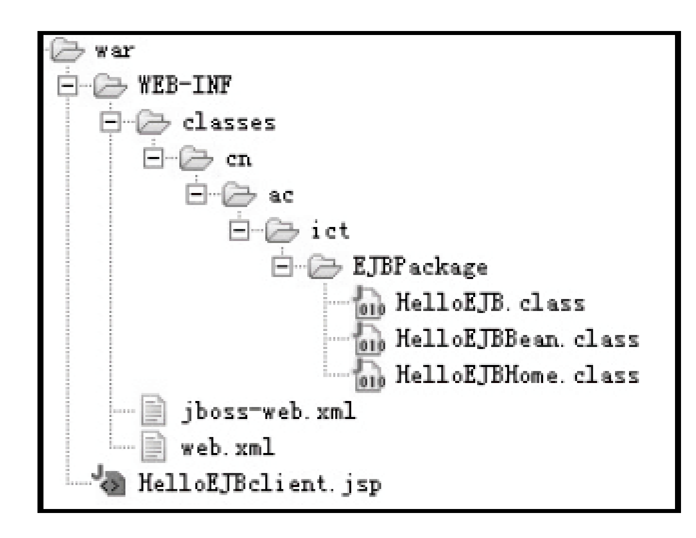


图 28.4　Web 应用文件结构

3．给 Web 应用打包并验证其正确性

把相关的文件组织好后，文件的结构如图 28.4 所示。

在 DOS 窗口中，转到 Web 应用所在的目录，运行如下命令：

```
jar cvf HelloEJBWAR.war *.*
```

在 Web 应用所在的目录下（这里是 war 文件夹）会得到 HelloEJBWAR.war 文件。把这个文件复制到<JBOSS_HOME>/server/default/deploy 下可以单独发布 Web 应用（首先必须保证 EJB 组件被正确地发布了，因为 Web 应用涉及对 EJB 的引用，如果 EJB 组件配置错误，Web 应用将无法获得对 EJB 的引用）。

28.1.6　在 JBoss 中发布运行 J2EE 应用

要发布这个 J2EE 应用，还需要一个 J2EE 应用的发布描述文件 application.xml，在这个文件中声明这个 J2EE 应用所包含的 Web 应用以及 EJB 组件。如下是该文件的内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
<display-name>HelloEJB</display-name>
<module>
<-- 在下面分别定义在这个 J2EE 应用中使用的 Web 组件-->
<web>
      <web-uri>HelloEJBWAR.war</web-uri>
      <context-root>/HelloEJB</context-root>
</web>
</module>
<module>
<-- 在下面分别定义在这个 J2EE 应用中使用的 EJB 组件-->
<ejb>HelloEJB.jar</ejb>
```

网络协议是什么，EJB 使用相同的编程 API 和语义以 Java RMI-IIOP 访问分布式对象。协议的细节对应用程序和 Bean 开发人员隐藏；对于所有供应商来说，定位和使用分布式 Bean 的方法是相同的。

注意： Enterprise Bean 与 JavaBeans 不同。JavaBeans 是使用 java.beans 包开发的，是 Java 2 标准版的一部分。JavaBeans 是一台计算机上同一个地址空间中运行的组件，它是进程内组件。Enterprise Bean 是使用 javax.ejb 包开发的，它是标准 JDK 的扩展，是 Java 2 Enterprise Edition 的一部分。Enterprise Bean 是在多台计算机上跨几个地址空间运行的组件，因此 Enterprise Bean 是进程间组件。JavaBeans 通常用作 GUI 窗口小部件，而 Enterprise Bean 则用作分布式商业对象。

28.2.2　实体 EJB

实体 Bean 是两种主要 Bean（实体和会话）中的一种。实体 Bean 用于表示数据库中的数据。它向 JDBC 或其他一些后端 API 经常访问的数据提供了一个面向对象的接口。不仅如此，实体 Bean 提供了一个组件模型，可以让 Bean 开发人员将精力集中在 Bean 的商业逻辑上，而容器负责管理持续、事务和访问控制。

实体 EJB 用在处理大量、并发的客户端请求的情况，它在实现业务逻辑的同时，作为数据库的一个缓冲。在服务量大的情况下，能减轻数据库的负担，提高业务处理能力。

实体 EJB 封装了业务逻辑实现，并且可以供多个客户使用。除了实现业务逻辑外，实体 EJB 的属性可以用来代表商业过程中处理的永久性数据。一个简单的实体 Bean 可以定义成代表数据库表的一个记录，也就是每一个实体对象代表一条具体的数据库记录。更复杂的实体 Bean 可以代表数据库表间关联视图。

有两种基本的实体 Bean：容器管理的持续（CMP）和 Bean 管理的持续（BMP）。容器使用 CMP 管理实体 Bean 的持续。供应商工具用于将实体字段映射到数据库，并且绝对没有数据库访问代码写入 Bean 类。使用 BMP，实体 Bean 包含了数据库访问代码（通常是 JDBC），负责读取其自身状态并将此状态写入数据库。BMP 实体对此有很大帮助，因为容器将提醒 Bean 何时需要更新状态或从数据库读取状态。容器还可以处理任何锁定或事务，因此数据库可以保持完整性。

部署在容器中实体 Bean 的 Home 接口的实现是由容器提供的。并且容器确保客户端能够通过 JNDI 访问部署在容器中的每个实体 Bean 的 Home 接口。实现实体 Bean Home 接口的对象是 EJBHome。

通过实体 Bean Home 接口，客户端可以进行如下操作：

- 创建新的实体对象。
- 查找存在的实体对象。
- 删除实体对象。
- 执行主逻辑方法。
- 获取主接口的句柄。

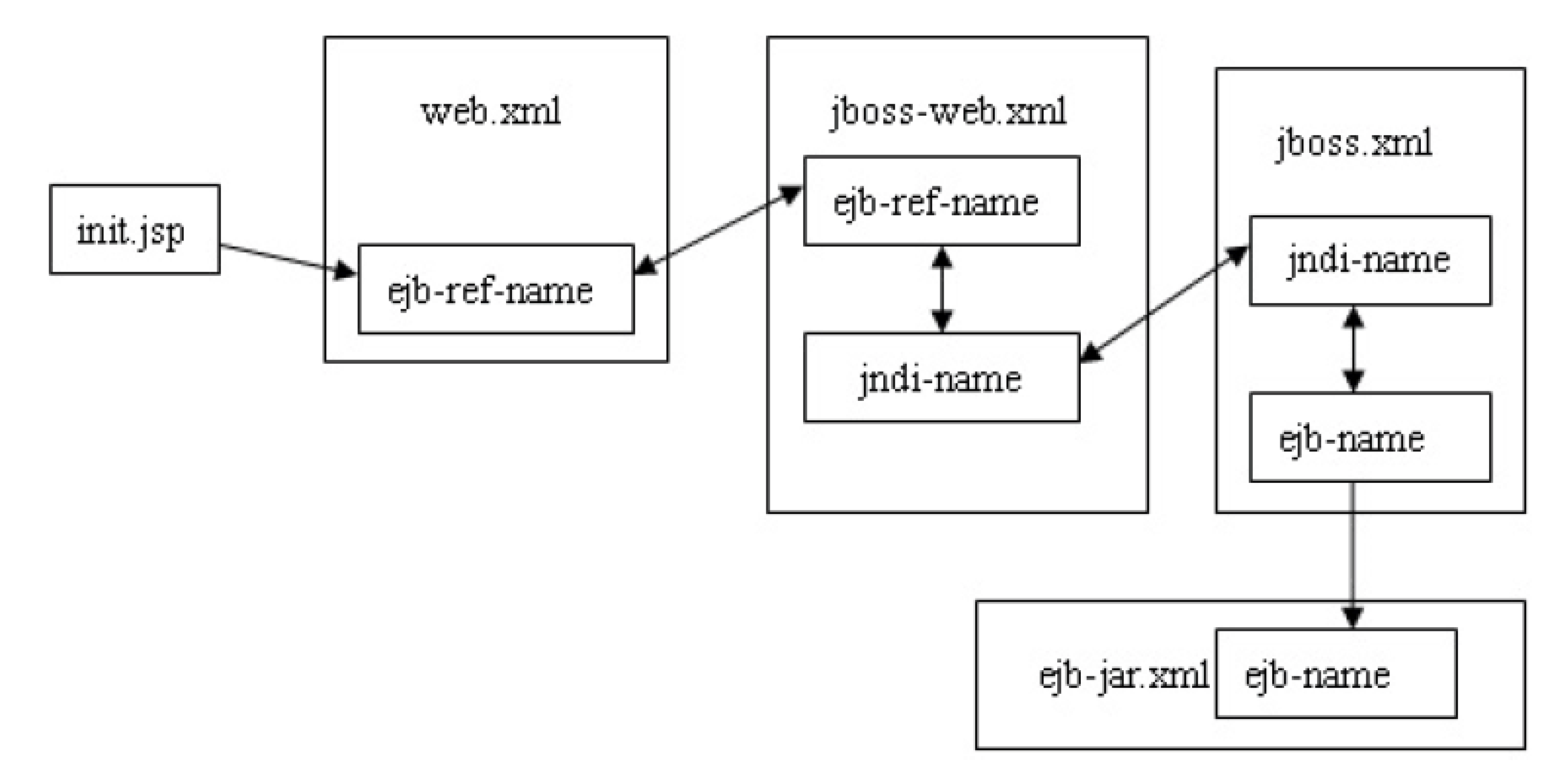


图 28.7　Web 应用中使用 EJB 时配置文件关系

28.3　网上书店——J2EE 应用实例

28.3.1　创建 EJB 组件

1．编写 Remote 接口文件

Remote 接口中定义了客户可以调用的业务方法，这些方法在 Enterprise Bean 类中会得到具体的实现，以下为 Remote 接口 BookShopDBEJBBean.java 的代码：

```
package cn.ac.ict.BookStoreOnline;

import java.rmi.RemoteException;
import java.util.ArrayList;

import javax.ejb.CreateException;
import javax.ejb.EJBObject;

public interface BookShopDBEJBBean extends EJBObject {
	//根据图书 ID 获取一个商品对象
	public Book getBookbyBid(String pid) throws RemoteException,CreateException;
	//使用指定的 SQL 语句查询数据库，并返回所有的图书对象集合
	public ArrayList getBooks(String sql) throws RemoteException,CreateException;

}
```

可以看到在 Remote 接口中定义了两个方法 getBookbyBid 和 getBooks，这里只作声明，不提供具体实现。

2．编写 Home 接口文件

Home 接口定义了创建、查找和删除 EJB 的方法。在本例中只有一个 create 方法，该方法返回一个 BookShopDBEJBBean 对象（Remote 接口类型）的远程引用。下面是 Home 接口 BookShopDBEJBHome.java 的代码：

```
        books = new ArrayList();
    }
    //使用指定的 SQL 语句查询数据库，并返回所有的商品对象集合
    public ArrayList getBooks(String sql){
        ResultSet rs = null;
        books.clear();
        try{
            Statement stmt = conn.createStatement();
            rs = stmt.executeQuery(sql);
            while(rs.next()){
                Book ptemp = new Book();
                ptemp.setBookid(rs.getString("bookid"));
                ptemp.setBookname(rs.getString("bookname"));
                ptemp.setPress(rs.getString("press"));
                ptemp.setAuthor(rs.getString("author"));
        ptemp.setPublishdate((java.util.Date)rs.getDate("publishdate"));
        ptemp.setSellDate((java.util.Date)rs.getDate("selldate"));
                ptemp.setRealprice(rs.getFloat("realprice"));
                ptemp.setCutprice(rs.getFloat("cutprice"));
                ptemp.setPicture(rs.getString("pic"));
                ptemp.setAmount(rs.getInt("amount"));
                ptemp.setHit(rs.getInt("hit"));
                ptemp.setDescription(rs.getString("description"));
                books.add((Object)ptemp);
            }
            stmt.close();
        }catch(Exception e){

        }
        return books;
    }
    //根据商品 ID 获取一个商品对象
    public Book getBookbyBid(String bookid){
        return (Book) getBooks("select * from books where bookid="+bookid).get(0);

    }

    public void ejbActivate() throws EJBException, RemoteException {
    }

    public void ejbPassivate() throws EJBException, RemoteException {
    }

    public void ejbRemove() throws EJBException, RemoteException {
        try{
            //关闭数据库连接
                conn.close();
            }catch(SQLException se){
```

```
<!--- 定义一个无状态的会话 Bean --->
                    <session-type>Stateless</session-type>
                    <transaction-type>Container</transaction-type>
               </session>
          </enterprise-beans>
</ejb-jar>
```

上面发布描述文件声明了一个无状态的会话 Bean（session-type 元素指定），并指明其 Remote 接口（remote 元素指定）、Home 接口（home 元素指定）和 Enterprise Bean 类（ejb-class 元素指定）对应的 Java 类文件。

要在 JBoss 上发布 EJB 组件还需要一个 jboss.xml 文件，它是只有在 JBoss 上发布 EJB 组件时才需要的文件，在其他的 EJB 容器中发布时就需要替换成其他的文件，比如 WebLogic 使用 weblogic-ejb-jar.xml 文件，在这个文件中为 EJB 指定 JNDI 名字，下面是文件的内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>bookstore</ejb-name>
      <jndi-name>ejb/bookstorejb</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

5．发布 EJB 组件并验证其正确性

到目前为止，EJB 组件的相关文件都已经准备好了，对 Java 类文件进行编译后，把相关文件组织好后，程序的结构如图 28.8 所示，然后给 EJB 组件打包并发布。

在 DOS 窗口中，转到 EJB 组件所在的目录（图 28.8 中为 ejb 文件夹），然后运行如下命令：

```
jar vcf bookshop.jar *.*
```

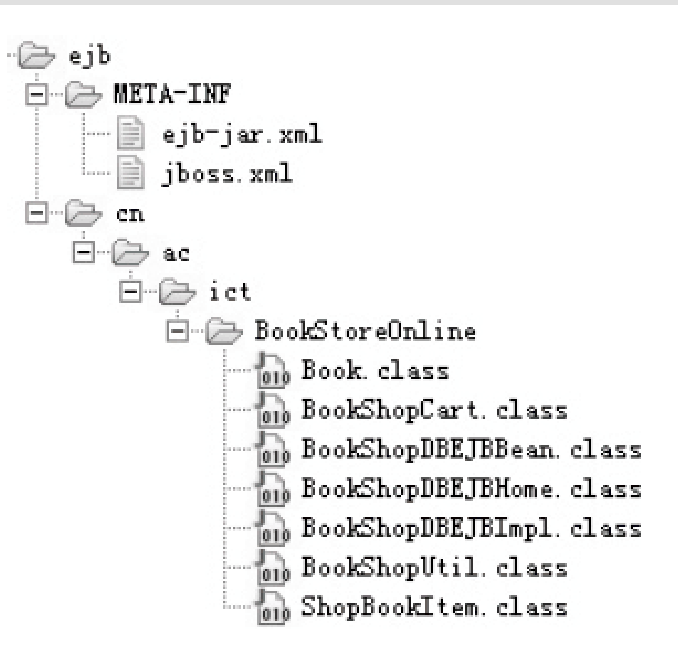


图 28.8　EJB 组件文件结构

此时，在该目录下会生成 bookshop.jar 文件。单独发布此 EJB 组件时把这个 JAR 文件复制到<JBOSS_HOME>/server/default/deploy 下就可以了，如果此 EJB 组件的相关配置都正确，JBoss 启动时会提示正确，否则需要根据 JBoss 提供的错误信息对类文件或配置文件进行修改。

引用可以通过包含这个文件的方式获得。这个文件被命名为 init.jsp（可以随意取），下面是这个文件的内容：

```
<%@ page language="java" errorPage="error.jsp" pageEncoding="GB2312" %>
<%@ page import="javax.ejb.*,javax.naming.*,javax.rmi.*"%>
<%@ page import="cn.ac.ict.BookStoreOnline.*"%>
<%@ page import="java.util.*"%>
<%!
    private BookShopDBEJBBean bookstore;
    public void jspInit(){
    bookstore = (BookShopDBEJBBean)getServletContext().getAttribute("bookstore");
    if(bookstore == null){
    try{
//得到初始化上下文
    InitialContext ic = new InitialContext();
//查找到 EJB 的引用
    Object objRef = ic.lookup("java:comp/env/ejb/bookstorejb");
    BookShopDBEJBHome  home  =  (BookShopDBEJBHome)PortableRemoteObject.narrow
(objRef,cn.ac.ict.BookStoreOnline.BookShopDBEJBHome.class);
//创建一个 EJB 的实现类对象
    bookstore = home.create();
//把这个类对象设置为 ServletContext 范围的属性
    getServletContext().setAttribute("bookstore",bookstore);
    }catch(Exception re){
    System.out.print(re);
    }//
    }//if
    }//init

    public void jspDestroy(){
    bookstore = null;
    }

    public String convert(String s){
    try{
    return new String(s.getBytes("ISO-8859-1"),"GB2312");
    }catch(Exception e){
    return null;
    }
    }
%>
```

从上面的代码中可以看到如下这段代码通过查找 JNDI 名获得该 JNDI 资源的远程引用：

```
//得到初始化上下文
    InitialContext ic = new InitialContext();
//查找到 EJB 的引用
    Object objRef = ic.lookup("java:comp/env/ejb/bookstorejb");
```

然后把这个对象转化为 BookShopDBEJBHome 类型：

```
<application>
<display-name>bookshop</display-name>
<module>
  //下面分别定义在这个 J2EE 应用中使用的 Web 模型
  <web>
     <web-uri>bookshop.war</web-uri>
     <context-root>/bookstore</context-root>
</web>
</module>
     <module>
     //下面分别定义在这个 J2EE 应用中使用的 EJB 模型
<ejb>bookshop.jar</ejb>
</module>
</application>
```

以上代码指明该 J2EE 应用中包含了一个 shoponlineWeb 应用，其 WAR 文件为 bookshop.war，访问路径是/bookstore；包含一个 EJB 组件，其 JAR 文件是 bookshop.jar。

把该实例需要的文件组织后，文件的结构如图 28.10 所示。



图 28.10　J2EE 应用文件结构

在 DOS 窗口中，转到 J2EE 应用的目录，运行以下命令：

```
jar vcf bookshop.ear *.*
```

此时，在这个目录下将生成 bookshop.ear 文件。这样就可以按照如下步骤发布 J2EE 应用了：

（1）把 MySQL 数据库的驱动程序复制到<JBOSS_HOME>/server/default/lib 目录下。

（2）把 bookshop.ear 文件复制到<JBOSS_HOME>/server/default/deploy 目录下。

（3）启动 MySQL 服务器。

（4）运行<JBOSS_HOME>/bin/run.bat，启动 JBoss 和 Tomcat 服务器。

（5）访问 http://localhost:8080/bookshop/list.jsp，将会看到这个 J2EE 应用的所有商品列表页面。

28.4　小　　结

EJB 技术和 JSP 技术共同成为 J2EE 技术的两大方面，EJB 技术使得多层结构的应用系统的开发变得容易，是企业级计算的重要技术，EJB 技术和 JSP 技术的结合使用也是很重要的技术手段，可以使读者在阅读本章的基础上进一步了解 EJB 技术。

本章首先从一个简单的例子出发介绍了 EJB 技术和 JSP 技术的结合使用，而后简单介绍了一下 EJB 技术，最后以一个实际的网上书店购物车的例子演示了实际中如何使用它们，读者应该尝试开发一个系统，才会更好地理解它们之间是如何结合起来的。

第 29 章　JSP 作为客户访问 CORBA 服务

CORBA 是用于分布式计算的成熟技术，在实际应用中得到了广泛的关注。它与 Web 应用的结合也是很重要的一个方面。在本章中介绍一些 CORBA 的基础知识，并讲述如何实现 CORBA 服务以及如何在 Tomcat 中配置，使得 CORBA 技术和 Web 开发结合起来应用。

29.1　快速体验 CORBA——使用 JSP 访问 CORBA 的简单例子

29.1.1　CORBA 简介

随着因特网技术的日益成熟，学校、商业企业和很多宽带用户正享受着高速、低价网络信息传输所带来的高品质数字生活。但随着网络规模的不断扩大以及计算机软硬件技术水平的大幅提高，传统的应用软件系统的实现方式面临着巨大的挑战。

首先，在企业级应用中，硬件系统集成商基于性能、价格、服务等方面的考虑，通常在同一系统中集成来自不同厂商的硬件设备、操作系统、数据库平台和网络协议，由此带来的异构性给应用软件的互操作性、兼容性以及平滑升级能力带来了严重问题。另外，随着基于网络的业务不断增多，传统的客户/服务器（C/S）模式的分布式应用方式越来越显示出在运行效率、系统网络安全性和系统升级能力等方面的局限性。

为了解决分布式计算环境（Distributed Computing Environment，DCE）中不同硬件设备和软件系统的互联，增强网络间软件的互操作性，解决传统分布式计算模式中的不足等问题，对象管理组织（OMG）提出了公共对象请求代理体系结构（CORBA），以增强软件系统间的互操作能力，使构造灵活的分布式应用系统成为可能。

正是基于面向对象技术的发展和成熟、客户/服务器软件系统模式的普遍应用以及集成已有系统等方面的需求，推动了 CORBA 技术的成熟与发展。作为面向对象系统的对象通信的核心，CORBA 为当今网络计算环境带来了真正意义上的互联。

CORBA（Common Object Request Broker Architecture）是由 OMG 提出的应用软件体系结构和对象技术规范，其核心是一套标准的语言、接口和协议，以支持异构分布应用程序间的互操作及平台无关的编程语言的对象重用。

OMG 于 1990 年初步提出了 CORBA 思想，到现在的最新版本是 3.0 版本。

```
        public void muti(int a, int b, IntHolder c) {
                c.value=a*b;
        }

        public void div(int a, int b, IntHolder c) {
                c.value=a/b;
        }

        public void sub(int a, int b, IntHolder c) {
                c.value=a-b;
        }
}
```

其中，各个方法涉及的 c 变量是 a 和 b 的运算结果，都使用了 IntHolder 类型，用于返回结果。

3．编写服务端实现

下面是服务器 CalServer.java 的内容：

```
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

import ArithApp.calculate;
import ArithApp.calculateHelper;

public class CalServer {

        public CalServer() {
                super();
        }

        public static void main(String args[]) {
               try{
                 //创建并初始化 ORB
                 ORB orb = ORB.init(args, null);

                 //创建一个接口实现的例子，并把它向 ORB 注册
                 CalImpl impl = new CalImpl(orb);

                 //获得 RootPOA  的引用并激活 POAManager
                 POA rootpoa = POAHelper.narrow(
                    orb.resolve_initial_references("RootPOA"));
                 rootpoa.the_POAManager().activate();

                 //从服务中得到对象的引用
                 org.omg.CORBA.Object ref =
```

```
                NamingContextExtHelper.narrow(objRef);

            //从命名上下文中获取接口实现对象
            String name = "Cal";
            calculate impl = calculateHelper.narrow(ncRef.resolve_str(name));

            System.out.println("Handle   obtained on server object: " + impl);

            int a =90;
            int b =7;
            IntHolder c = new IntHolder();
            // 调用乘法
            impl.muti(a,b,c);
            // 输出结果
            System.out.println("The result of a×b is: "+c.value);
            impl.div(a,b,c);
            //输出结果
            System.out.println("The result of a÷b is: "+c.value);
            impl.add(a,b,c);
            //输出结果
            System.out.println("The result of a＋b is: "+c.value);
            impl.sub(a,b,c);
            //输出结果
            System.out.println("The result of a－b is: "+c.value);

        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}
```

5．运行程序

把所有的文件都编写并编译完成后，各个 Java 源文件的存放位置如图 29.1 所示，生成的字节码文件放在 bin 目录下，与 Java 源文件的字节码文件在 bin 中的相对位置和在 src 中相同。按照下面的步骤执行该应用程序：

（1）启动 orbd，在应用程序根目录 CorbaApp 下运行：

```
orbd -ORBInitialPort 3588
```

（2）启动服务器端，在应用程序根目录 CorbaApp 下运行：

```
java CalServer -ORBInitialPort 3588
```

（3）这时可以看到命令行提示服务器运行成功并等待客户端调用。

（4）启动客户端，在应用程序根目录 CorbaApp 下运行：

```
java CalClient -ORBInitialPort 3588
```

客户端成功运行就可以看到如图 29.2 所示的效果。

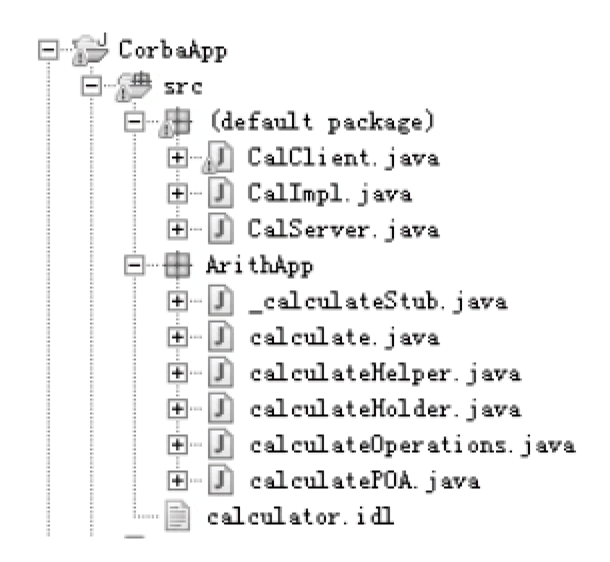


图 29.1　CORBA 应用目录结构图

```
C:\WINDOWS\System32\cmd.exe
E:\CorbaApp\bin>java CalClient -ORBInitialPort 3588
Handle  obtained on server object: ArithApp._calculateStub:IOR:000000000000001b4
9444c3a417269746841707 02f63616c63756c6174653a312e30000000000000010000000000006c0
00102000000000d3139322e3136382e312e31390000074100000021afabcb00000000209b70a1000
0000001000000000000000000000000040000000000a0000000000000001000000010000002000000000
00100010000000020501000100010020000101090000000001000101 00
The result of a×b is: 630
The result of a÷b is: 12
The result of a+b is: 97
The result of a-b is: 83

E:\CorbaApp\bin>
```

图 29.2　CORBA 应用效果

29.2　CORBA 技术构成

CORBA 体系结构中设计的几个核心概念有：对象请求代理、接口定义语言、对象适配器等，在本节中将结合 CORBA 的体系结构介绍 CORBA 架构中的一些核心组件和概念。

29.2.1　对象请求代理（Object Request Broker，ORB）

CORBA 体系结构的核心是 ORB，简单而言，ORB 就是使客户应用程序能调用远端对象方法的一种机制。

当客户端的应用程序需要调用远程对象的某个方法时，首先需要得到这个远程对象的引用，这样才可以像调用本地对象的方法一样调用远程对象。当客户端发出一个调用请求时，ORB 会截获这个请求（通过客户 Stub 完成），因为客户和服务器有可能在不同的网络、不同的操作系统上，甚至用不同的语言实现，ORB 还要负责将调用的名字、参数等编码成标准的方式（称为 Marshaling），然后通过网络传输到服务器方（即使在同一台机器上也要这样进行），并通过将参数 Unmarshaling 的过程，传到正确的对象上（这整个过程称为重定向，Redirecting），服务器对象完成方法调用后，ORB 通过同样的 Marshaling/Unmarshaling 方式将结果返回给客户。

下面简单介绍一下 ORB 的结构，图 29.3 说明的是客户端发送一个请求到对象的实现过程。

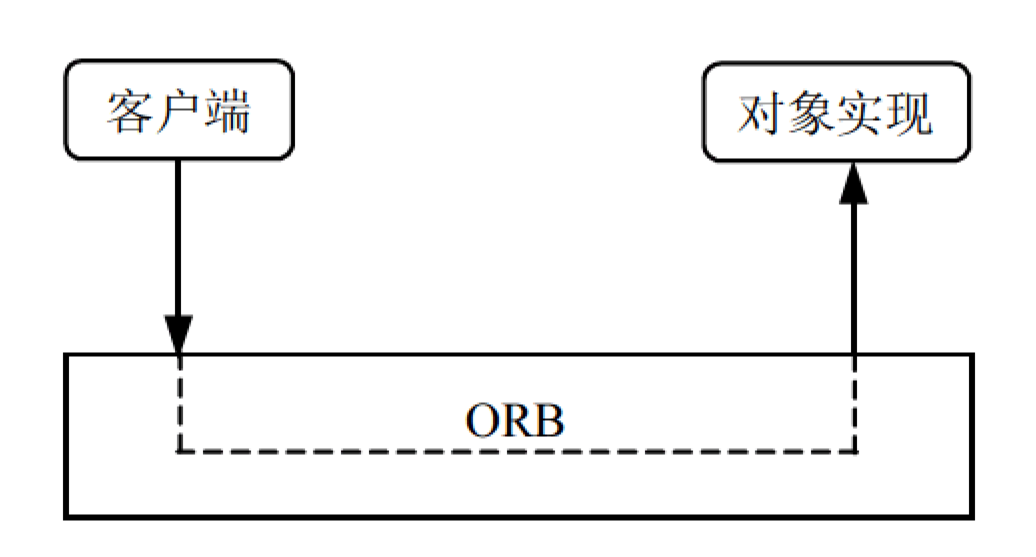


图 29.3　客户端发送一个请求到对象的实现

客户端是希望对某对象执行操作的实体。对象的实现是通过使用一段代码和数据来实现。ORB 负责实现下面的必要功能：对象定位（对该请求找到对象的实现）、确信服务器端能接受请求、把客户请求重新定位到服务器端的对象实现上并让对象的实现准备好接受请求、交换数据。客户端的接口完全独立于对象的位置，其实现的语言等因素也不影响对象接口的实现。图 29.4 展示的是一个独立的 ORB 的结构。

29.2.3　接口仓库（Interface Repository，IR）

接口仓库，顾名思义可以理解为 CORBA 分布式对象接口定义的集中存储区，可以用来辅助用户使用、发布、管理这个存储区中记录的对象接口。CORBA 是分布计算的模型，所以这里所说的存储区也不一定是某个文件或某个数据库，它也是分布的，这里的集中存储是逻辑上的存储，具体的存放位置可能在不同的计算机上。

接口仓库要实现的主要功能有：

- 为不同 ORB 之间的互操作提供实现。
- 供 ORB 用来检查请求签名。
- 供 ORB 检查接口之间大继承关系。

接口仓库提供的功能及工具主要依赖于 CORBA 提供商，根据不同的工具，接口仓库还可以实现更多的功能。

注意： ORB 是对象请求代理（Object Request Broker）的简写，是使客户应用程序能调用远端对象方法的一种机制。

29.2.4　对象适配器（Object Adapter，OA）

对象适配器是 ORB 的一部分，它帮助 ORB 把请求传给对象并激活该对象，一个对象适配器只能与一个对象实现联系，它可以被定义来支持特定的对象实现类型（如 OODB 对象适配器用于持续对象，而 library 对象适配器用于非远程对象）。对象适配器的作用主要有：

- 产生和解释对象引用。
- 方法调用。
- 相互作用的安全性。
- 对象的激活实现及撤销实现。
- 把对象引用映射到相应的对象实现。
- 注册对象实现。

29.2.5　动态调用接口（Dynamic Invocation Interface，DII）和静态调用接口（Static Invocation Interface，SII）

动态调用接口和静态调用接口都位于客户端，它们负责发送客户端的调用请求。

在客户端，ORB 使用动态调用接口（Dynamic Invocation Interface，DII）来发送操作调用；在服务器端，OA 通过动态框架接口（Dynamic Skeleton Interface，DSI）来传输一个操作调用，它是服务器端对应 DII 的行为。动态调用接口（DII）可以和动态骨架接口（DSI）结合，使得客户可以在不知道服务器对象的接口的情况下调用服务器对象。

在客户端，客户与ORB之间的静态接口被称为静态调用接口(Static Invocation Interface，SII)，在服务器端，与这个接口对应的被称为静态框架接口（Static Skeleton Interface，SSI）。

29.2.6　GIOP 和 IIOP

GIOP 是一种通信协议，它是客户方的 ORB 和服务器方的 ORB 通信的协议，它规定了客户和服务器 ORBs 间的通信机制。

GIOP 是 CORBA 方法调用的核心部分。GIOP 不基于任何特别的网络协议，如 IPX 或 TCP/IP。为了确保互操作性，OMG 必须将 GIOP 定义在所有供应商都支持的特定传输之上。因此，OMG 在最广泛使用的通信传输平台——TCP/IP 上标准化 GIOP。GIOP 加 TCP/IP 就是 IIOP。

29.3　股票查询服务——CORBA 服务实例

29.3.1　使用 IDL 语言定义 IDL 接口并编译映射到 Java 程序

这是开发 CORBA 应用程序的第一步，接口中定义了客户端应用程序可以访问的方法，但没有具体的实现。文件的具体内容如下（文件名为 Stock.idl）：

```
module StockApplication {
    interface StockOperation {
            void getStockName(in string StockID,out string StockName);
            void getStockCurrentPrice(in string StockID,out float currentprice);
            void getStockDiff(in string StockID,out float diff);

    };
};
```

这个例子是要实现股票的查询功能，在上面的接口中定义了 3 个方法，分别用于根据股票代码获取股票的名称、当前的价格和浮动状况。保存文件到 src 文件夹后，在 src 目录下执行：

```
idlj –fall calculator.idl
```

可以看到在目录下生成了 ArithApp 文件夹，下面有 6 个文件，分别如下：

- StockOperationPOA.java
- StockOperationOperations.java
- StockOperationHolder.java
- StockOperationHelper.java
- StockOperation.java
- _StockOperationStub.java

至于各个文件的作用这里就不深入介绍了，读者可以在深入学习 CORBA 的基础上了解它们的用途。

29.3.2　实现 IDL 接口

在 29.3.1 节中的 IDL 文件中定义了几个方法，虽然也生成了对应的 Java 文件，但并没有实际执行的代码，下面是实现这个接口的代码：

```
import org.omg.CORBA.IntHolder;
import org.omg.CORBA.ORB;
import org.omg.CORBA.*;
import java.sql.*;

import StockApplication.StockOperationPOA;

public class StockQueryImpl extends StockOperationPOA {
	private ORB orb;
	private boolean connected = false;
	private String defaultSQLdriver = "com.mysql.jdbc.Driver";
	private String defaultmySQLURL = "jdbc:mysql://localhost/StockBase?user=root&password
=ict";
	private Connection conn;
	public StockQueryImpl(ORB orb){
		this.orb =orb;
	}

	private void connectDB(String mySqlDriver,String SQLurl){
		try{
		  //加载数据库驱动程序
		   Class.forName(mySqlDriver);
		   }catch(Exception e){
		   System.out.println("Driver Error");
		   }

		try{
			//与数据库建立连接，得到 Connection 的对象
			conn=DriverManager.getConnection(SQLurl);
			connected = true;
			}catch(Exception e){
			System.out.println("无法连接");
			}
		}

		public void getStockName(String StockID, StringHolder StockName){
			if(!connected){
				connectDB(defaultSQLdriver,defaultmySQLURL);
			}
			try{
				//得到 Statement 的对象
				Statement stmt=conn.createStatement();
				ResultSet rs=null;
```

法中都要先判断是否具有数据库连接，如果没有就要调用 connectDB 方法连接数据库：

```
		if(!connected){
			connectDB(defaultSQLdriver,defaultmySQLURL);
		}
```

注意： 由于这个应用中需要连接 MySQL 数据库，需要把数据库驱动程序放到类路径中。

29.3.3　编写服务端实现

服务器端的主要功能包括：

- 创建并初始化 ORB。
- 创建一个接口实现的例子，并把它向 ORB 注册。
- 获得 RootPOA 的引用并激活 POAManager。
- 从服务中得到对象的引用。
- 从命名服务中获取根命名上下文并把这个新对象注册为“Cal”。
- 等待客户端的调用。

下面是类 StockQueryServer 的代码：

```
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

import StockApplication.StockOperationPOA;
import StockApplication.StockOperation;
import StockApplication.StockOperationHelper;

public class StockQueryServer {

	public StockQueryServer() {
		super();
	}

	public static void main(String args[]) {
		try{
		  //创建并初始化 ORB
		  ORB orb = ORB.init(args, null);
		  //创建一个接口实现的例子，并把它向 ORB 注册
		  StockQueryImpl impl = new StockQueryImpl(orb);
		  //获得 RootPOA  的引用并激活 POAManager
		  POA rootpoa = POAHelper.narrow(
		     orb.resolve_initial_references("RootPOA"));
		  rootpoa.the_POAManager().activate();
		  //从服务中得到对象的引用
```

样的：

- ❑ 创建并初始化 ORB。
- ❑ 获得根命名上下文的引用。
- ❑ 寻找名为“Stock”的对象，并获得其引用。
- ❑ 调用接口实现中实现的方法。

在本实例中，使用一个 Servlet 首先从服务器端获取一个远程对象，并把这个对象作为 Application 范围的属性，在其他页面需要使用时只需要从 JSP 的 application 隐含对象中获取这个属性就可以了，下面是获取远程对象的 Servlet：

```
package cn.ac.ict;

import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import StockApplication.StockOperation;
import StockApplication.StockOperationHelper;

import javax.servlet.http.*;

public class StockQueryClientServlet extends HttpServlet   {

      /**
       * 构造方法
       */
      public StockQueryClientServlet() {
            super();
      }

      public void init() {
            try {
             String args[] = new String[2];
             args[0] = "-ORBInitialPort";
             args[1] = "3588";

               // 创建并初始化 ORB
               ORB orb = ORB.init(args, null);
               // 获得根命名上下文
               org.omg.CORBA.Object objRef =
                  orb.resolve_initial_references("NameService");
               // Use NamingContextExt instead of NamingContext. This is
               // part of the Interoperable Naming Service.
               NamingContextExt ncRef =
                  NamingContextExtHelper.narrow(objRef);
               // 在命名服务中处理对象引用
               String name = "Stock";
               StockOperation impl = StockOperationHelper.narrow(ncRef.resolve_str(name));
              this.getServletContext().setAttribute("StockQuery",impl);

            } catch (Exception e) {
```

```
                        StockOperation impl = (StockOperation)this.getServletContext().getAttribute
("StockQuery");
                        StringHolder sname = new StringHolder();
                        FloatHolder currentprice = new FloatHolder();
                        FloatHolder diff = new FloatHolder();
//调用查询数据的方法
                        impl.getStockName(SID,sname);
                        impl.getStockCurrentPrice(SID,currentprice);
                        impl.getStockDiff(SID,diff);

                    out.write("\r\n");
                    out.write("<table width=\"580\" border=\"0\" cellspacing=\"0\" cellpadding=\"0\"
align=\"center\">\r\n");
                    out.write("<thead align=\"center\">查询结果</thead>\r\n");
                    out.write("   <tr>\r\n");
                    out.write("      <td>股票代码</td>\r\n");
                    out.write("      <td>股票名称</td>\r\n");
                    out.write("      <td>当前价格</td>\r\n");
                    out.write("      <td>浮动</td>\r\n");
                    out.write("   </tr>\r\n");
                    out.write("   <tr>\r\n");
                    out.write("      <td>");
                    out.print( SID );
                    out.write("</td>\r\n");
                    out.write("      <td>");
                    out.print( sname.value );
                    out.write("</td>\r\n");
                    out.write("      <td>");
                    out.print( currentprice.value );
                    out.write("</td>\r\n");
                    out.write("      <td>");
                    out.print( diff.value );
                    out.write("</td>\r\n");
                    out.write("   </tr>\r\n");
                    out.write("</table>\r\n");
              }
                    out.write("\r\n");
                    out.write("</body>\r\n");
                    out.write("</html>\r\n");
      }
}
```

由于使用了 Servlet 获取远程对象和调用业务方法，就需要在 web.xml 文件中配置这些 Servlet，配置的代码将在 29.3.6 节介绍。

29.3.6　配置 CORBA 服务的 Servlet 客户端

配置 CORBA 服务的 Servlet 客户端与配置普通的 Servlet 没有什么区别，在发布描述文

（4）启动 Tomcat 服务器，在浏览器地址栏中输入如下地址：

```
http://localhost:8080/32/stockQuery.stockquery
```

可以看到页面显示如图 29.6 所示。

如果查询了数据库中不存在的股票，就会显示出错误的股票名称，一个简单的可能页面如图 29.7 所示。而输入的股票代号存在时就会显示出正确的信息，页面效果如图 29.8 所示。

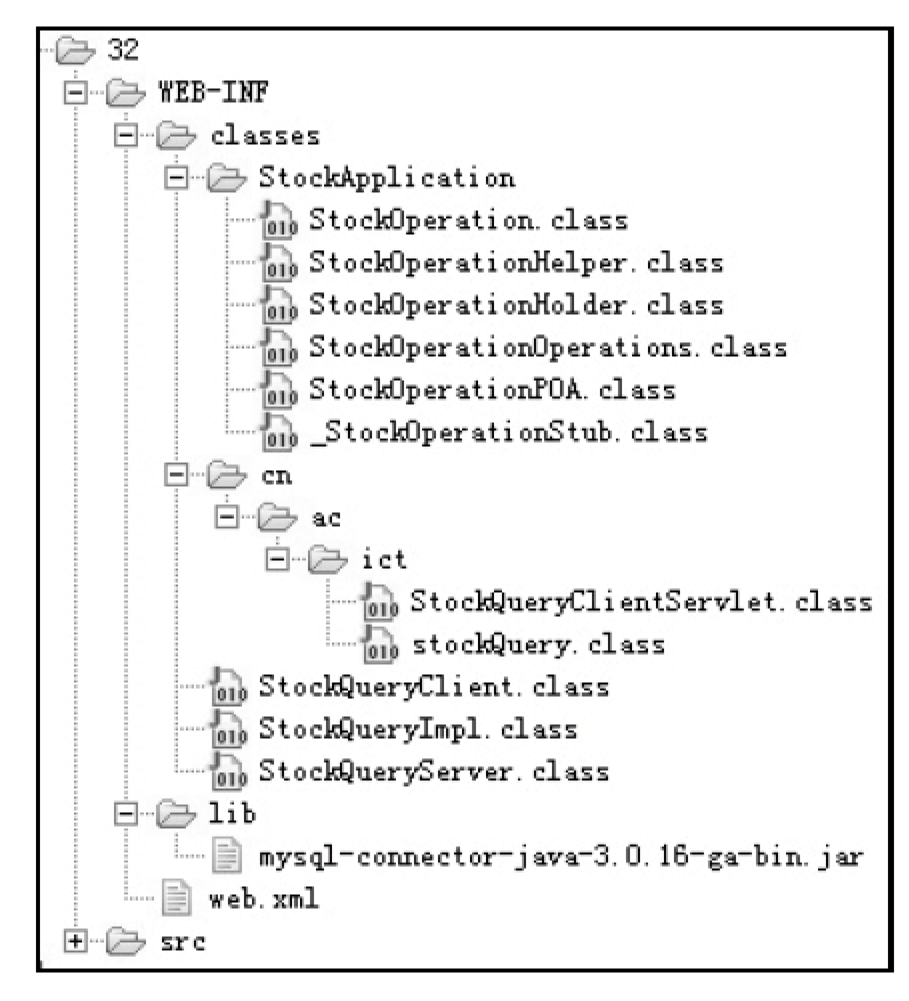


图 29.5　Web 应用目录结构



图 29.6　输入页面

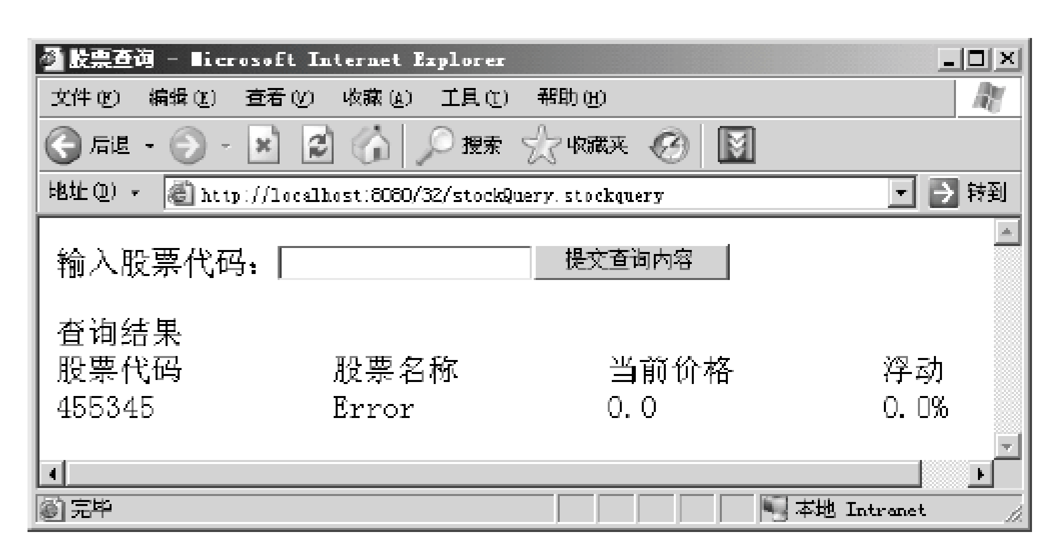


图 29.7　查询结果出现错误



图 29.8　查询结果正确显示

29.4　小　　结

本章简单介绍了在分布式计算领域得到广泛应用的 CORBA 技术。CORBA 是支持异构分布应用程序间的互操作及平台无关的，使用它可以开发性能良好的、分布式的、平台无关的应用程序。

本章先通过一个四则运算的例子演示了 CORBA 技术与 Java 技术的结合，然后解释了 CORBA 的一些组件和重要概念，但可以说这只是会常常遇到的一部分，读者要掌握 CORBA 还需要参考更多的资料，在最后一节演示了一个实际使用 CORBA 的股票查询例子，读者应该对照例子更好地理解如何使用 CORBA。

第 30 章　Velocity 模板语言

Velocity 的主要用途就是简化 Web 应用的开发，是一个功能强大的模板工具。在本章中将会通过一个简单的例子首先告诉读者如何使用这个模板工具，之后会介绍 Velocity 模板语言的相关知识。

30.1　Velocity 入门

30.1.1　简介

Velocity 是一个基于 Java 的模版引擎，是 Apache 软件组织提供的一项开放源代码的项目。它允许 Web 页面设计者通过 Velocity 模板语言定义模板，在模板中不需要加入任何 Java 代码，由 Java 开发者编写程序来设置上下文（包含用户替换模板中某个代码的数据），Velocity 引擎就可以把模板和上下文结合起来生成动态的网页。这样 Web 设计者可以根据 MVC 模式和 Java 程序员并行工作，Web 设计者可以单独专注于设计良好的站点，而程序员则可单独专注于编写底层代码。Velocity 将 Java 代码从 Web 页面中分离出来，使站点在长时间运行后仍然具有很好的可维护性，并提供了一个除 JSP 和 PHP 之外的可行的被选方案。

Velocity 模板语言（VTL）为网页开发者提供了简捷的方法来生成动态网页。即使没有编程经验的人也可以很快地掌握 VTL 语言。VTL 模板中不包含任何 Java 代码，是其与 JSP 网页的一个重要区别，而且 VTL 模板不需要经过 JSP 编译器的编译，VTL 模板的解析是由 Velocity 完成的。

Velocity 可用来从模板产生 Web 页面、SQL、PostScript 以及其他输出。也可用于一个独立的程序以产生源代码和报告，或者作为其他系统的一个集成组件，不过它的主要用途还是简化 Web 开发。

下面运行一个简单的 Velocity 例子，了解开发基于 Velocity 的程序需要的必要步骤。

30.1.2　安装 Velocity

从 Velocity 的主页（http://jakarta.apache.org/velocity/index.html）上下载 Velocity-1.4 版本（velocity-1.4.zip）。把这个压缩文件解压缩，在解压的目录下可以看到两个 JAR 文件：velocity-1.4.jar 和 velocity-dep-1.4.jar。这就是开发基于 Velocity 的程序需要的最关键的两个 JAR 文件。只要把这两个 JAR 文件复制到 Web 应用程序的 WEB-INF 目录下的 lib 文件夹下就可以完成安装了。

```
        if(path==null){
            System.out.println("HelloVelocity.loadConfiguration():"+
            "unable to get the current webapp root. Using '/'. ");
            path="/";
            }
//设置属性
            p.setProperty(Velocity.FILE_RESOURCE_LOADER_PATH,path);
            p.setProperty("runtime.log",path+"velocity.log");

        return p;
    }
    //对客户请求进行处理，把模板中的变量替换为动态数据
    public Template handleRequest(HttpServletRequest request, HttpServletResponse response,
Context context){
        Template outty=null;
        try{
        String username = "rambler";
        Date today = new Date();
//把模板中的变量替换为动态数据
        context.put("name",username);
        context.put("date",today.toString());
        outty=getTemplate("hellovelocity.vm");
        }catch(ParseErrorException ex1){
            System.out.println("HelloVelocity: parse error for template "+ex1);
            }catch(ResourceNotFoundException ex2){
                System.out.println("HelloVelocity: template not found "+ex2);
                }catch(Exception ex3){
                    System.out.println("HelloVelocity: error "+ex3);
                    }
        return outty;
        }
    }
```

在 HelloVelocity.java 中对 VelocityServlet 的两个重要方法进行了重写，下面分别介绍它们完成的工作：

- loadConfiguration()方法是 VelocityServlet 的初始化方法，它由 VelocityServlet.init()调用，并设置模板所在的路径，默认情况下就是 Web 应用的根目录，如果使用 WAR 文件发布应用在 Tomcat 上仍然可以运行。
- handleRequest()方法是用于提供服务的方法，也就是完成从模板生成动态网页的过程，它由 VelocityServlet 自动调用。

在 HelloVelocity 的 handleRequest 方法中指定了用户的名字为“rambler”，并生成了一个日期对象，然后对模板上下文中的变量 name 和 date 进行了替换：

```
        String username = "rambler";
        Date today = new Date();
        context.put("name",username);
        context.put("date",today.toString());
```

最后它获得模板，并返回代表 hellovelocity.vm 模板的对象：

```
outty=getTemplate("hellovelocity.vm");
……
return outty;
```

3．配置 HelloVelocity

要使得应用能够识别这个 Servlet，需要在 web.xml 文件中添加如下元素：

```
<servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>cn.ac.ict.HelloVelocity</servlet-class>
   </servlet>

   <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <url-pattern>/hello</url-pattern>
   </servlet-mapping>
```

4．发布基于 Velocity 的 Web 应用

上面的这个应用完成之后其程序的结构如图 30.1 所示。

也可以按照如下步骤发布这个应用：

（1）编译 HelloVelocity.java，需要把 servelet-api.jar 和 velocity-1.4.jar 文件放到类路径中。

（2）把编译生成的类文件复制到 Web 应用的 WEB-INF\classes\cn\ac\ict 目录下。

（3）把 hellovelocity.vm 文件放到 Web 应用的根目录下。

（4）把 velocity-dep-1.4.jar 文件和 velocity-1.4.jar 文件复制到 WEB-INF\lib 目录下。

（5）启动 Tomcat，然后在浏览器地址栏中输入如下地址：http://localhost:8080/velocity/hello，可以看到页面显示如图 30.2 所示。

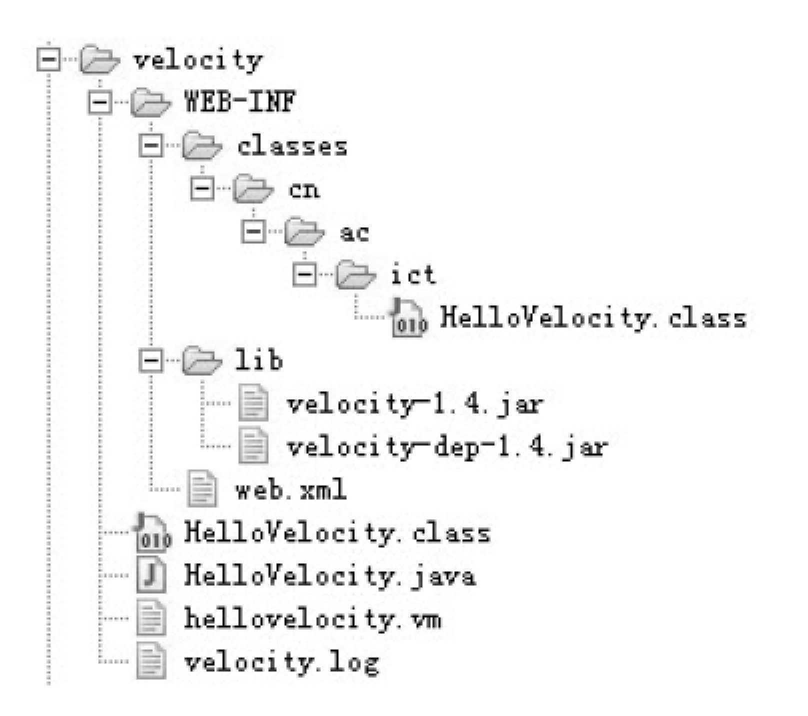


图 30.1　基于 Velocity 的 Web 应用的程序结构



图 30.2　基于 Velocity 的 Web 应用效果

30.2　注　　释

VTL 支持单行注释（以##开始）和多行注释（包括在#*和*#之间）。例如：

```
## This is a single line comment.
```

30.3.2　属性引用

属性应用的使用格式是：$VTL 标识符. VTL 标识符。

下面是属性引用的例子：

```
$customer.Address
$purchase.Total
```

第一个引用$customer.Address 可以看作名为 customer 的 Hashtable 对象中键值为 Address 的值，也可以看作是一个方法：$customer.getAddress()。属性引用可以有两种解释方式，Velocity 会根据开发者编写的程序在运行时决定如何解释。那什么时候按方法解释，什么时候按照 Hashtable 对象解释呢？

其实属性引用的两种解释方式对应了其两种赋值方式，当属性引用使用 Hashtable 对象赋值时就会按照 Hashtable 对象的方式解释，当属性引用在 JavaBeans 中使用某个方法赋值时就需要被解释成方法。下面分别举例说明。

1．使用 Hashtable 对象赋值

使用 Hashtable 对象赋值，Velocity 引擎就会把属性引用按照 Hashtable 对象的键值引用，下面是一个使用 Hashtable 对象赋值的例子。

Velocity 模板如下：

```
<html>
<title>Hello Velocity</title>
<body>
<h3>Product Info are listed below:</h3><br/>
<p>Product Name:$product.name     </p><br/>
<p>Product Price:$product.price     </p><br/>
<p>Product Company:$product.comp </p>       <br/>

</body>
</html>
```

对应的 VelocityServlet 如下：

```
package cn.ac.ict;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.velocity.*;
import org.apache.velocity.context.*;
import org.apache.velocity.servlet.*;
import org.apache.velocity.app.*;
import org.apache.velocity.exception.*;
```

```
            <servlet-name>product_1</servlet-name>
            <url-pattern>/product_1</url-pattern>
    </servlet-mapping>
```

然后把编译后的类文件复制到 WEB-INF\classes\cn\ac\ict 目录下，启动 Tomcat，在浏览器地址栏中输入：http://localhost:8080/velocity/product_1，可以看到这时的页面显示如图 30.3 所示。

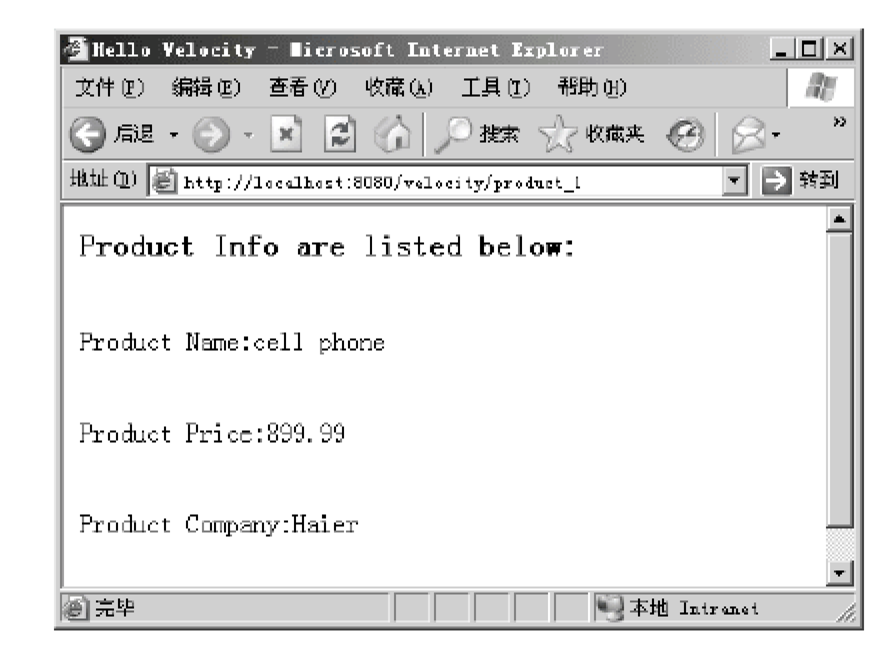


图 30.3　属性引用的效果

2．在 JavaBeans 中用方法赋值

当属性引用在 JavaBeans 中使用某个方法赋值时就需要被解释成方法。下面是一个使用 JavaBeans 中用方法赋值的例子，其中模板仍然使用上面的模板，下面是对应的 VelocityServlet 的源代码：

```
package cn.ac.ict;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.velocity.*;
import org.apache.velocity.context.*;
import org.apache.velocity.servlet.*;
import org.apache.velocity.app.*;
import org.apache.velocity.exception.*;

public class ProductVelocity_2 extends VelocityServlet{
//加载配置信息
  protected Properties loadConfiguration(ServletConfig config)
            throws IOException,FileNotFoundException{
      Properties p=new Properties();
      String path=config.getServletContext().getRealPath("/");
      if(path==null){
            System.out.println("ProductVelocity.loadConfiguration():"+
            "unable to get the current webapp root. Using '/'. ");
            path="/";
            }
//设置属性
            p.setProperty(Velocity.FILE_RESOURCE_LOADER_PATH,path);
            p.setProperty("runtime.log",path+"velocity.log");

      return p;
  }
  //对客户请求进行处理，把模板中的变量替换为动态数据
  public Template handleRequest(HttpServletRequest request, HttpServletResponse response,
Context context){
      Template outty=null;
      try{
//把模板中的变量替换为动态数据
```

30.3.4　正式引用符（Formal Reference Notation）

在上面几节中讲述的都是引用符的简略形式，除了上面介绍的引用符，还有一种正式引用符，它比简略形式的引用在标识符外多了一个括号，例如下面的例子都是正式的应用符：

```
${mudSlinger}
${customer.Address}
${purchase.getTotal()}
```

其实简略形式的引用符和正式引用符效果是一样的，所以一般情况下都使用简略引用符，不过在一些情况下就必须使用正式引用符以区分引用符和普通的字符串。例如：

```
Jack is a $vicemaniac
```

如果刚好有一个名为 vice 的变量，上面的代码解释起来就会出现问题，是应该把 vicemaniac 作为引用的标识符还是把 vice 作为标识符呢？为了避免系统运行的这种不确定性，就可以使用正式引用符来避免问题的发生，例如，上面的例子，实际上要引用名为 vice 的变量，为了避免误解，就可以像下面这样写：

```
Jack is a ${vice}maniac
```

这样 Velocity 就会知道是把 vice 作为变量进行引用，而不会去引用 vicemaniac，避免了错误的发生。

30.3.5　安静引用符（Quiet Reference Notation）

下面的例子：

```
<input type="text" name="email" value="$email"/>
```

初始时，$email 没有值，所以文本框中会显示值$email，但实际应用中它显示空白更合适一些，而不是这样一个客户无法识别的符号。要解决这个问题，就可以使用安静引用符来避免这种情况的发生了，可以按照如下方式修改：

```
<input type="text" name="email" value="$!email"/>
```

可见安静引用符就是比常规引用符在$后多了一个!。这样，如果变量 email 没有值，文本框中会显示空白，而不是$email 了。

正式引用符可以和安静引用符一起使用，如下：

```
<input type="text" name="email" value="$!{email}"/>
```

30.4　指　　令

模板设计者可以使用上面介绍的引用输出动态网页的内容，可以使用指令操纵 Java 代

码的输出，从而控制页面的外观和内容。Velocity 提供了丰富的 Java 代码操作指令，下面分别介绍各个指令的作用和使用方法。

30.4.1　#set 指令——变量赋值

格式：#set(LHS = RHS)。

- ❑ LHS 可以是变量引用或属性引用。
- ❑ RHS 可以是引用、字符串、数字、ArrayList 或 Map。

#set 指令用来为应用变量和引用属性赋值，下面是一些使用#set 指令的正确例子：

```
#set( $primate = "monkey" )
#set( $customer.Behavior = $primate )
#set( $monkey = $bill ) ## variable reference
#set( $monkey.Friend = "monica" ) ## string literal
#set( $monkey.Blame = $whitehouse.Leak ) ## property reference
#set( $monkey.Plan = $spindoctor.weave($web) ) ## method reference
#set( $monkey.Number = 123 ) ##number literal
#set( $monkey.Say = ["Not", $my, "fault"] ) ## ArrayList
```

在上面的例子中演示了把允许的数据类型赋值给引用的例子。对于 ArrayList 和 Map，可以使用对应的 Java 方法访问其中的元素值：

```
$monkey.Say.get(0)
$monkey.Map.get("bannana")
$monkey.Map.banana ## same as above
```

RHS 可以是简单的算术表达式，例如：

```
#set( $value = $foo + 1 ) ## Addition
#set( $value = $bar - 1 ) ## Subtraction
#set( $value = $foo * $bar ) ## Multiplication
#set( $value = $foo / $bar ) ## Division
#set( $value = $foo % $bar ) ## Remainder
```

注意：（1）算术表达式只支持整型。/的结果为整数；如果为非整型数值，返回 null。
（2）如果 RHS 的结果为 null，是不会赋值给 LHS 的。

看下面的例子：

```
#set( $criteria = ["name", "address"] )
#foreach( $criterion in $criteria )
    #set( $result = $query.criteria($criterion) )
    #if( $result )
        Query was successful
    #end
#end
```

上面使用$result 检查是否执行成功是有问题的。如果第一次执行成功，$result 不为 null，则后面的执行不论是否成功，检查条件总是成立。改进的方法是在每次执行前初始化为 false：

```
#set( $criteria = ["name", "address"] )
#foreach( $criterion in $criteria )
#set( $result = false )
    #set( $result = $query.criteria($criterion) )
    #if( $result )
        Query was successful
    #end
#end
```

注意：上例中使用了一些尚未介绍的指令，读者可以参考后面几节的介绍。

String 文字可以使用双引号或单引号括起来。两者的主要区别是双引号中的引用会替换成相应的值，而单引号中的引用原样输出。

```
#set( $directoryRoot = "www" )
#set( $templateName = "index.vm" )
#set( $template = "$directoryRoot/$templateName" )
$template
```

输出结果是：

```
www/index.vm
```

如果使用单引号：

```
#set( $template = '$directoryRoot/$templateName' )
```

输出结果是：

```
$directoryRoot/$templateName
```

使用双引号可以实现字符串的串联，如下面的例子：

```
#set( $size = "Big" )
      #set( $name = "Ben" )
#set($clock = "${size}Tall$name" )
      The clock is $clock.
```

30.4.2　#if/#elseif/#else 指令——条件语句

Velocity 的#if 指令在条件成立时，显示#if 和#end 之间的内容，否则显示#else 和#end 之间的内容。下面是一个例子：

```
#if( $foo )
   <strong>Velocity!</strong>
#end
```

Velocity 会先对变量$foo 求值，确定条件是否成立，条件成立的条件有两个：

- 如果$foo 是 boolean，则$foo 要为 true。
- 否则，$foo 不为 null。

而条件不成立的条件就是上面的对立面了：

- $foo 是 boolean，并且$foo 值是 false。

```
#foreach( $customer in $customerList )
    <tr><td>$velocityCount</td><td>$customer.Name</td></tr>
#end
</table>
```

循环次数的默认变量的名称和初始值都保存在 velocity.properties 文件中，默认是$velocityCount，默认初始值是 1，可以修改这个文件，使其为 0 或者 1，以适应不同人的编程习惯。velocity.properties 文件的定义如下：

```
# Default name of the loop counter
# variable reference.
directive.foreach.counter.name = velocityCount

# Default starting value of the loop
# counter variable reference.
directive.foreach.counter.initial.value = 1
```

下面举一个使用#foreach 指令的例子，它把一个 Hashtable 中所有的对象信息输出，这个例子使用的模板 products.vm 如下：

```
<html>
<title>Hello Velocity</title>
<body>
<h3>All Product Info are listed below:</h3><br/>
<table border=1>
<tr>
<td>Name</td>
<td>Price</td>
<td>Company</td>
</tr>
#foreach($product in $productlist)
<tr>
<td>$product.name </td>
<td>$product.price</td>
<td>$product.comp </td>
</tr>
#end

</table>

</body>
</html>
```

对应的 VelocityServlet 的源代码如下：

```
package cn.ac.ict;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import org.apache.velocity.*;
import org.apache.velocity.context.*;
import org.apache.velocity.servlet.*;
import org.apache.velocity.app.*;
import org.apache.velocity.exception.*;

public class ProductsVelocity extends VelocityServlet{
//加载配置信息
  protected Properties loadConfiguration(ServletConfig config)
             throws IOException,FileNotFoundException{
      Properties p=new Properties();
      String path=config.getServletContext().getRealPath("/");
      if(path==null){
             System.out.println("ProductVelocity.loadConfiguration():"+
             "unable to get the current webapp root. Using '/'. ");
             path="/";
             }
             p.setProperty(Velocity.FILE_RESOURCE_LOADER_PATH,path);
             p.setProperty("runtime.log",path+"velocity.log");

      return p;
  }
  //对客户请求进行处理，把模板中的变量替换为动态数据
  public Template handleRequest(HttpServletRequest request, HttpServletResponse response,
Context context){
      Template outty=null;
      try{
      Hashtable productlist = new Hashtable();
      Product product = new Product();
      product.setName("cell phone");
      product.setPrice("899.99");
      product.setComp("Haier");
      productlist.put(product.getName(),product);

      product = new Product();
      product.setName("air conditioning");
      product.setPrice("8899.99");
      product.setComp("Haier");
      productlist.put(product.getName(),product);

      product = new Product();
      product.setName("clock");
      product.setPrice("99.99");
      product.setComp("Haier");
      productlist.put(product.getName(),product);

      context.put("productlist",productlist);
      outty=getTemplate("products.vm");
      }catch(ParseErrorException ex1){
```

```
            System.out.println("ProductVelocity: parse error for template "+ex1);
            }catch(ResourceNotFoundException ex2){
                System.out.println("ProductVelocity: template not found "+ex2);
                }catch(Exception ex3){
                    System.out.println("ProductVelocity: error "+ex3);
                    }
        return outty;
        }
  }
```

在这个 VelocityServlet 中建立一个 Hashtable 对象，并向其中添加 3 个 Product 对象，这样，在页面中就可以循环这个 Hashtable 对象中所有的对象，然后输出各个对象的信息。下面是所使用的 JavaBeans 类 Product.java 的源代码：

```
package cn.ac.ict;
  public class Product {
    private String name;
    private String price;
    private String comp;
    //下面是各个属性的设置方法
    public void setName(String nn){
        name = nn;
    }
    public void setPrice(String pp){
        price = pp;
    }
    public void setComp(String cc){
        comp = cc;
    }
//下面是各个属性的获取方法
    public String getName(){
        return name;
    }
    public String getPrice(){
        return price ;
    }
    public String getComp(){
        return comp;
    }
}
```

把上面的文件编译，按照上面介绍的方法发布，注意在 web.xml 文件中添加如下元素：

```
<servlet>
    <servlet-name>products</servlet-name>
    <servlet-class>cn.ac.ict.ProductsVelocity</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>products</servlet-name>
```

#parse 指令与#include 指令的区别就在于#include 指令只是把文件包含到指令所在的位置，而#parse 指令是把被包含文件解析后的结果包含到指令所在的地方。

30.4.6　#stop 指令——停止模板执行

#stop 指令使得设计者可以中止模板的执行并返回，这对于调试来说是很有用的。

30.4.7　#macro 指令——定义 VTL 模板

#macro 指令允许定义一段重复使用的 VTL 模板（称为 Veloci macros）。Velocimacros 可以有 0 或多个参数。下面是一个例子：

```
#macro( tablerows $color $somelist )
#foreach( $something in $somelist )
    <tr><td bgcolor=$color>$something</td></tr>
#end
#end
```

这个称为 tablerows 的 Velocimacros 带两个参数：color（$color）和 array（$somelist）。下面的代码包含对 tablerows 的调用：

```
#set( $greatlakes = ["Superior","Michigan","Huron","Erie","Ontario"] )
#set( $color = "blue" )
<table>
    #tablerows( $color $greatlakes )
</table>
```

输出结果为：

```
<table>
    <tr><td bgcolor="blue">Superior</td></tr>
    <tr><td bgcolor="blue">Michigan</td></tr>
    <tr><td bgcolor="blue">Huron</td></tr>
    <tr><td bgcolor="blue">Erie</td></tr>
    <tr><td bgcolor="blue">Ontario</td></tr>
</table>
```

Velocimacros 可以在 VTL 模板中定义为 inline，这样对其他的 VTL 模板是无效的；要使 Velocimacros 在所有 VTL 模板中共享，可以将 Velocimacros 定义在 Velocimacros 模板库（全局）中。

Velocimacros 属性在 velocity.properties 文件中定义，提供实现 Velocimacros 的灵活性：

- velocimacro.library = VM_global_library.vm。
- velocimacro.permissions.allow.inline = true。
- velocimacro.permissions.allow.inline.to.replace.global = false。
- velocimacro.permissions.allow.inline.local.scope = false。
- velocimacro.context.localscope = false。

30.5　VTL 的其他特征

30.5.1　关系运算和逻辑运算

Velocity 使用“＝”判断两个变量是否相等，下面是一个演示等号运算符如何使用的例子：

```
#set ($foo = "deoxyribonucleic acid")
#set ($bar = "ribonucleic acid")

#if ($foo == $bar)
  In this case it's clear they aren't equivalent. So...
#else
  They are not equivalent and this will be the output.
#end
```

这时，$foo 和$bar 是不相等的，输出的结果是：

```
They are not equivalent and this will be the output.
```

Velocity 也有 AND、OR 和 NOT 运算符，其中 AND 使用的符号是&&，下面是一个使用逻辑与的例子：

```
## logical AND

#if( $foo && $bar )
    <strong> This AND that</strong>
#end
```

#if 指令的条件只有在$foo 和$bar 都为 true 的情况下才返回 true，如果$foo 为 false，Velocity 引擎将不会继续求$bar 值，因为任何一个为 false，表达式为 false；如果$foo 为 true，才会继续求$bar 的值。

逻辑 OR 的操作是类似的，它使用的符号是||，下面是一个使用逻辑或的例子：

```
## logical OR

#if( $foo || $bar )
     <strong>This OR That</strong>
#end
```

如果$foo 为 true，Velocity 引擎将不会继续求$bar 值，而直接设置表达式为 true，逻辑 NOT 的符号是！，它只有一个参数，下面是一个逻辑非的例子。

```
##logical NOT

#if( !$foo )
  <strong>NOT that</strong>
#end
```

这里的 m 和 n 都必须是整数。m 和 n 的大小没有限制，如果 n 大于 m，计数从大到小；如果 n 小于 m，计数从小到大。下面举几个范围操作的例子：

第 1 个例子：

```
#foreach( $foo in [1..5] )
$foo
#end
```

上面的代码输出：

```
1 2 3 4 5
```

第 2 个例子：

```
#foreach( $bar in [2..-2] )
$bar
#end
```

这个例子的输出结果是：

```
2 1 0 -1 -2
```

第 3 个例子：

```
#set( $arr = [0..1] )
#foreach( $i in $arr )
$i
#end
```

这个例子的输出结果是：

```
0 1
```

第 4 个例子：

```
[1..3]
```

这个例子的输出结果是：

```
[1..3]
```

30.6　小　　结

Velocity 是一个基于 Java 的模版引擎。它使用模板生成动态网页，并可以作为 WebWork 的视图组件，其使用者越来越多。创建基于 Velocity 的 Web 应用包括两个步骤：

- 创建模板。
- 创建扩展 VelocityServlet 的 Servlet 类。

在本章中介绍了 Velocity 的大量基础知识，还举了大量的例子来讲述各个指令或者语法的工作过程，读者可以重点学习 Velocity 的基础知识，之后就会发现把 Velocity 和 Tomcat 结合不是件难事了。

第 31 章　JSP 案例：在线图书订购系统

本章继续介绍一个实际应用的例子。这是一个简单的在线图书定购系统，它使用了 JavaBeans、JDBC 数据库连接等技术。通过这个例子，读者应该能够很好地理解很多商业应用的设计，对以后从事开发工作是很有意义的。

31.1　BookStore 实例介绍

31.1.1　BookStore 的结构描述

这个 Web 应用由 3 个 JavaBeans、5 个 JSP 页面和 1 个工具类构成。其中各个构成元素的作用如下：

（1）商品基本信息的 JavaBeans——Book，这个 JavaBeans 用于保存图书的基本信息，如书名、价格、出版社等信息。

（2）购物车 Bean——BookShopCart，这个 JavaBeans 用于模仿一个购物车，在这个购物车中可以加入很多的图书项目。

（3）购买项目 Bean——ShopBookItem，这个 JavaBeans 表示一个购买项目，这个购买项目是指多个同一图书。例如客户购买了 10 本《Tomcat Web 开发及整合应用》，则这 10 本书就构成了一个购买项目，购买项目封装了图书的信息和图书的数量信息。

（4）数据库连接初始化——init.jsp，这个 JSP 文件用于获取数据库连接，它没有任何显示任务，任何需要数据库连接的页面都必须包含这个文件。

（5）图书列表页面——list.jsp，这个 JSP 页面用于列出所有客户可以购买的图书信息，读者在这个页面中可以选择某个图书将其加入到购物车中。

（6）添加图书确认页面——add.jsp，这个 JSP 页面用于显示客户想要加入购物车的图书信息，在这个页面中，客户可以修改购买的数量，并提交把图书加入到购物车中。

（7）中转页面——putintocart.jsp，这个 JSP 页面没有显示任务，它的功能就是获得客户要购买的某种商品的详细信息，并把商品的详细信息封装成 Book 对象，把图书信息对象保存到 Session 中，最后把客户请求转发到添加图书确认页面。

（8）查看购物车——showcart.jsp，这个 JSP 页面用于显示客户的购物车中的内容，并根据购物车的不同状态，显示不同的信息。

（9）工具类——ShopUtil.java，这个工具类被设计成提供一些有用的功能。在本实例中它只有一个方法，用于转换字符的编码方式，以支持显示中文。

31.1.2　BookStore 的功能介绍

在了解了 BookStore 的结构后，下面介绍一下 BookStore 的功能。它的功能可以通过图 31.1 来解释。

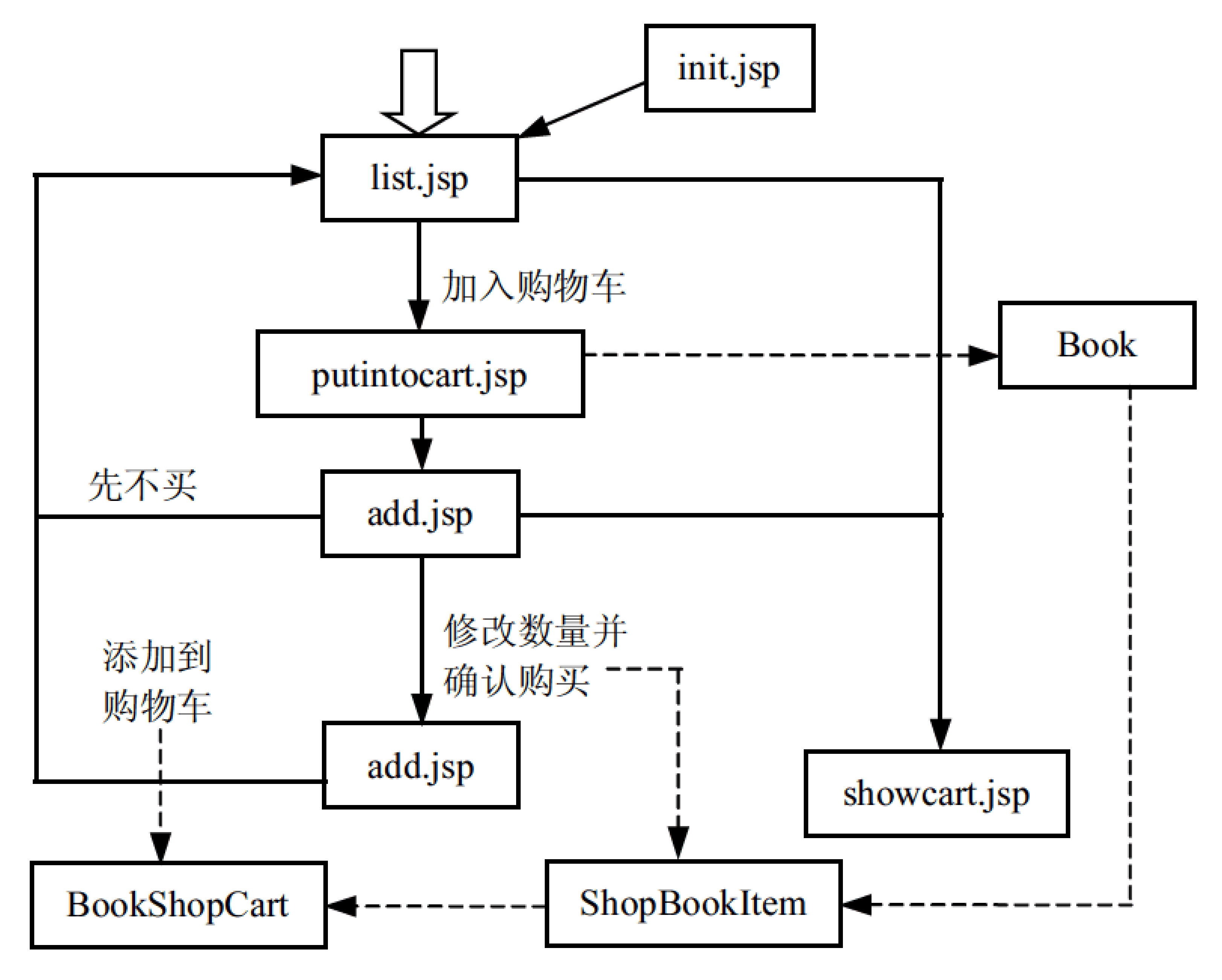


图 31.1　ShopOnLine 的功能介绍

客户访问这个 Web 应用的过程如下：

（1）客户登录这个 Web 应用后，先访问 list.jsp 页面，这个页面包含了 init.jsp，连接数据库，获得所有图书信息并在页面上显示。

（2）在 list.jsp 页面中，客户单击了某种图书后面的加入购物车链接后，转入 putintocart.jsp 页面，在这个页面中根据图书的 ID 获取图书的全部信息，并构造一个 Book 对象，并把它保存为 Session 范围的属性，然后把请求转发到 add.jsp。

（3）在 add.jsp 页面中显示客户刚刚选中的图书信息，客户修改购买的数量后提交给自身页面。

（4）此时的页面传送了一个名为 confirm 的属性，并把它设置为 true，如果 confirm 的值为 true，则把这个图书的信息加入到购物车中。首先构造一个 ShoppingItem 对象，然后把这个 ShoppingItem 对象加入到 ShoppingCart 对象中，并把请求转发到 list.jsp。

（5）在 add.jsp 页面中显示客户刚刚选中的图书信息，客户如果不决定购买，则可以单击“先不买这个商品”链接，会返回图书列表页面 list.jsp。

（6）在 add.jsp 页面和 list.jsp 页面中都提供了“查看购物车”链接，单击该链接，可以链接到 showcart.jsp 页面，在这个页面中显示客户购买的所有商品，并计算所有商品实际需要花费的金额以及节省的金额。

```
	 * @return Returns the bookid.
	 */
	public String getBookid() {
		return bookid;
	}

	/**
	 * @param bookname The bookname to set.
	 */
//书名的设置和获取方法
	public void setBookname(String bookname) {
		this.bookname = bookname;
	}

	/**
	 * @return Returns the bookname.
	 */
	public String getBookname() {
		return bookname;
	}

	/**
	 * @param author The author to set.
	 */
//作者的设置和获取方法
	public void setAuthor(String author) {
		this.author = author;
	}

	/**
	 * @return Returns the author.
	 */
	public String getAuthor() {
		return author;
	}

	/**
	 * @param publishdate The publishdate to set.
	 */
//出版日期的设置和获取方法
	public void setPublishdate(Date publishdate) {
		this.publishdate = publishdate;
	}

	/**
	 * @return Returns the publishdate.
	 */
	public Date getPublishdate() {
		return publishdate;
```

```
	}

	/**
	 * @param realprice The realprice to set.
	 */
//销售价的设置和获取方法
	public void setRealprice(float realprice) {
		this.realprice = realprice;
	}

	/**
	 * @return Returns the realprice.
	 */
	public float getRealprice() {
		return realprice;
	}

	/**
	 * @param cutprice The cutprice to set.
	 */
//会员价的设置和获取方法
	public void setCutprice(float cutprice) {
		this.cutprice = cutprice;
	}

	/**
	 * @return Returns the cutprice.
	 */
	public float getCutprice() {
		return cutprice;
	}

	/**
	 * @param amount The amount to set.
	 */
//本书的库存量的设置和获取方法
	public void setAmount(int amount) {
		this.amount = amount;
	}

	/**
	 * @return Returns the amount.
	 */
	public int getAmount() {
		return amount;
	}
	/**
	 * @param description The description to set.
	 */
```

```
//图书描述信息的设置和获取方法
        public void setDescription(String description) {
                this.description = description;
        }

        /**
         * @return Returns the description.
         */
        public String getDescription() {
                return description;
        }

        /**
         * @param hit The hit to set.
         */
//该书点击量的设置和获取方法
        public void setHit(int hit) {
                this.hit = hit;
        }

        /**
         * @return Returns the hit.
         */
        public int getHit() {
                return hit;
        }

        /**
         * @param sellDate The sellDate to set.
         */
//开始销售的日期的获取和设置方法
        public void setSellDate(Date sellDate) {
                this.sellDate = sellDate;
        }

        /**
         * @return Returns the sellDate.
         */
        public Date getSellDate() {
                return sellDate;
        }

        /**
         * @param press The press to set.
         */
//出版社的设置和获取方法
        public void setPress(String press) {
                this.press = press;
        }
```

```
        if(items.containsKey(bookid)){
            items.remove(bookid);
            ShopBookItem newitem = new ShopBookItem(book,num);
            items.put(bookid,newitem);
        }else{
            ShopBookItem newitem = new ShopBookItem(book,num);
            items.put(bookid,newitem);
            itemAmount++ ;
        }

    }
    /**
     * 从购物车中移除指定的图书项目
     */
    public synchronized void remove(String bookid){
        if(items.containsKey(bookid)){
            items.remove(bookid);
        }
    }
    //获取购物车中所有的图书项目
    public synchronized Collection getItems(){
        return items.values();
    }
    protected void finalize(){
        items.clear();
    }
    //获取当前购物车中图书项目的数目
    public synchronized int getItemAmount(){
        return itemAmount;
    }
    //计算购物车中所有图书的市场价值
    public synchronized float getTotalReal(){
        float total = 0.0F;
        for(Iterator it = getItems().iterator();it.hasNext();){
            ShopBookItem si = (ShopBookItem)it.next();
            Book book = si.getItem();
            total = book.getRealprice()*si.getAmount();
        }
        return total;
    }
    //计算购物车中在本站购物所需要的所有花费之和
    public synchronized float getTotalCut(){
        float total = 0.0F;
        for(Iterator it = getItems().iterator();it.hasNext();){
            ShopBookItem si = (ShopBookItem)it.next();
            Book book = si.getItem();
            total = book.getCutprice()*si.getAmount();
        }
        return total;
```

```
        }
        //清空购物车
        public void clear(){
                items.clear();
                itemAmount = 0;
        }

}
```

31.2.3　购买项目 Bean——ShopBookItem

购买项目 Bean——ShopBookItem.java 也是一个很简单的 JavaBeans。它在整个 Web 应用中的作用并不是很明显，它只是用来连接 Book 和 BookShopCart 的一个类，它的实现也不是很标准，但它可以完成封装 Book 的作用。下面是 ShopBookItem.java 的源代码：

```
package cn.ac.ict.BookStoreOnline;
import java.io.Serializable;
public class ShopBookItem implements Serializable {
        private Book item;
        private int amount = 0;

        public ShopBookItem() {
                super();
        }
        /**
         * 使用指定的图书和它的数目构造一个购买项目
         */
        public ShopBookItem(Book newbook,int num) {
                super();
                if(num>0){
                        item = newbook;
                        amount = num;
                }
        }
        public void InAmount(){
                amount++;
        }
        public void DeAmount(){
                amount--;
        }
//获取一个图书项目
        public Book getItem(){
                return item;
        }
//获取购物车中图书项目的数量
        public int getAmount(){
                return amount;
        }
}
```

31.4　JSP 页面

JSP 页面的主要作用就是要提供给客户一个良好的用户界面，同时也实现了一定的逻辑操作功能。

31.4.1　图书列表页面

实现图书列表的页面是 list.jsp 文件。在这个文件中，首先包含了用于连接数据库的 init.jsp 文件，获得数据库连接后，从中查询出所有的图书然后在页面上显示。下面是这个文件的源代码：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import="java.sql.*,cn.ac.ict.BookStoreOnline.*" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
    <title>图书列表</title>
  <style type="text/css">
  <!--
  .bookname {
      font-size: 20px;
      line-height: 30px;
      color: #CC9900;
  }
  .description {
      font-size: 13px;
      color: #FF0000;
      background-color: #CCCCCC;
  }
  .common {
      font-size: 14px;
      color: #333333;
  }
  -->
  </style>
</head>
<body bgcolor="#FFFFFF">
<%@ include file="init.jsp"%>
<%

      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select * from books");
%>
<table width="778" border="0" align="center">
```

```
}%>
</table>
</body>
</html>
```

在上面的代码中使用了一个工具类 ShopUtil，它只有一个很简单的静态方法用于把使用 ISO8859-1 编码的字符串转换为使用 gb2312 编码的字符串。本书不作介绍。

在本程序中使用如下语句获取数据库中的数据：

```
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select * from productlist");
```

然后就不断地循环，把所有商品的信息输出到页面显示。这个页面执行后，效果如图 31.2 所示。



图 31.2　所有可购买的图书列表

31.4.2　添加图书确认页面

在添加图书确认页面（add.jsp）中可以修改购买图书的数量，并确认把指定数量的图书加入到购物车中。同时，这个页面中表单的提交页面也是 add.jsp，根据传入的不同参数进行不同的操作。下面是 add.jsp 的源代码：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import = "java.sql.*,cn.ac.ict.BookStoreOnline.*"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
     <title>购买商品</title>
   <style type="text/css">
   <!--
   .top {
       font-size: 18px;
   }
.tabletop {
       font-size: 16px;
       color: #3991D0;
```

```
{
//如果客户对提交的信息取消了，将页面转发到 list.jsp
        session.removeAttribute(addpid);
        response.sendRedirect("list.jsp");
}
else{
%>
<form action='add.jsp' method='get'>
<table align=center>
   <thead><center>
   <p class="top">修改购买图书的数量并把它们加入到您的购物车</p>
</center></thead>
<tr>
   <td class="tabletop">图书名称</td>
   <td class="tabletop">出版社</td>
   <td class="tabletop">作者</td>
   <td class="tabletop">出版日期</td>
   <td class="tabletop">定价</td><td class="tabletop">会员价</td><td
class="tabletop">数量</td><td class="tabletop">小计</td></tr>
<tr>
<td
class="tablecontent"><%=BookShopUtil.getChinese(temp.getBookname())%></td>
<td
class="tablecontent"><%=BookShopUtil.getChinese(temp.getPress())%></td>
<td
class="tablecontent"><%=BookShopUtil.getChinese(temp.getAuthor())%></td>
<td class="tablecontent"><%=temp.getPublishdate()%></td>
<td class="tablecontent"><%=temp.getRealprice()%></td>
<td class="tablecontent"><%=temp.getCutprice()%></td>
<td class="tablecontent"><input type=text size=2 name="amount" value=<%=count%>></input>
</td>
<td class="tablecontent"><%=count*(temp.getRealprice())%></td>
</tr>
<input type="hidden" name="confirm" value="true">
<input type="hidden" name="add" value="<%=addpid%>">

<tr>

<td style="border:2 solid "  colspan=2 align=center><a href="list.jsp" class="linkstyle">先不买这个
商品</a></td>
<td style="border:2 solid "  colspan=3 align=center><a href="showcart.jsp"
class="linkstyle">查看购物车</a></td>
<td style="border:2 solid "  colspan=3 align=center><input type=submit size=2 name="buy"
value="确定购买"></input></td>
</tr>
</table>
</form>
<%}%>
</body>
</html>
```

这个页面的显示效果如图 31.3 所示。

图 31.3　添加图书确认页面

31.4.3　中转页面

在从图书列表页面到添加图书确认页面中间还有一个中转页面，它没有具体的显示任务，它完成一些处理任务后，把请求转发到其他页面。在本例中的 putintocart.jsp 就是这样一个页面。它完成的功能有：

- 获得客户要购买的某种图书的详细信息。
- 把图书的详细信息封装成 Product 对象。
- 把图书信息对象保存到 Session 中。
- 把客户请求转发到添加图书确认页面。

putintocart.jsp 的源代码如下：

```
<%@ page language="java" pageEncoding="GB2312" %>
<%@ page import = "java.sql.*,cn.ac.ict.BookStoreOnline.*"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
  <head>
    <title>Lomboz JSP</title>
  </head>
<body bgcolor="#FFFFFF">
<%@ include file="init.jsp"%>
<jsp:useBean id="shopcart" scope="session"
class="cn.ac.ict.BookStoreOnline.BookShopCart" />
<%
        String addpid = request.getParameter("add");
        if (addpid!=null){
//如果客户要添加一本图书，则从数据库中得到图书的详细信息
                Statement stmt = conn.createStatement();
                ResultSet rs = stmt.executeQuery("select * from books where bookid=' "+addpid+" ' ");
        rs.next();

        Book temp = new Book();
        temp.setBookid(addpid);
        temp.setBookname(rs.getString("bookname"));
        temp.setAuthor(rs.getString("author"));
```

```
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
  <head>
    <title>查看购物车</title>
  <style type="text/css">
  <!--
  .top {
      font-size: 18px;
  }
  .tabletop {
      font-size: 16px;
      color: #3991D0;
      background-color: #CCCCCC;
      border: 2px solid #666666;
}
.tablecontent {
      font-size: 14px;
      color: #000000;
      border-bottom-width: 1px;
      border-bottom-style: solid;
      border-bottom-color: #666666;
      text-align: center;
  }
  .linkstyle {
      text-decoration: none;
  }
  -->
  </style>
</head>
<body bgcolor="#FFFFFF">
<%@ include file="init.jsp" %>
<jsp:useBean id="shopcart" scope="session"
class="cn.ac.ict.BookStoreOnline.BookShopCart" />
<%
      String clear = request.getParameter("clear");
//客户请求清空购物车
      if(clear!=null&&clear.equalsIgnoreCase("true")){
            shopcart.clear();
%>
      <table align=center><tr><td class="top">您的购物车已经被清空，<a href = "list.jsp">返回图书
列表</a></td>
      </tr></table>
<%
      }
if(shopcart.getItemAmount()>0){
//列出购物车中所有的图书
%>
<table align=center border=0>
<thead align=center><h2 class="top">您在购物车中加入了如下图书商品</h2></thead>
<tr>
  <td class="tabletop">图书名称</td>
```

```
  <td class="tabletop">出版社</td>
  <td class="tabletop">作者</td>
  <td class="tabletop">出版日期</td>
  <td class="tabletop">定价</td><td class="tabletop">会员价</td><td
class="tabletop">数量</td><td class="tabletop">小计</td><td class="tabletop">修改</td></tr>
<%
Collection coll = shopcart.getItems();

Iterator ite = coll.iterator();
while(ite.hasNext()){
ShopBookItem si = (ShopBookItem)ite.next();
Book p = si.getItem();%>
<tr>
<td
class="tablecontent"><%=BookShopUtil.getChinese(p.getBookname())%></td>
<td class="tablecontent"><%=BookShopUtil.getChinese(p.getPress())%></td>
<td class="tablecontent"><%=BookShopUtil.getChinese(p.getAuthor())%></td>
<td class="tablecontent"><%=p.getPublishdate()%></td>
<td class="tablecontent"><%=p.getRealprice()%></td>
<td class="tablecontent"><%=p.getCutprice()%></td>
<td class="tablecontent"><%=si.getAmount()%></td>
<td class="tablecontent"><%=p.getRealprice()*si.getAmount()%></td>
<td class="tablecontent"><a href='add.jsp?amount=<%=si.getAmount()%>&add=<%=p.getBookid()
%>' class="linkstyle">修改数量</a></td>
</tr>

<%}%>
<tr align=left><td colspan=9>本次购物您<font color=red>本来</font>需要花费<font color=red><%
=shopcart.getTotalReal()%></font>
在本站购买需要<font color=red><%=shopcart.getTotalCut()%></font></td></tr>
<tr align=left><td colspan=9>在本站购物为您<font color=red>节省了<%=(shopcart.getTotalReal()
-shopcart.getTotalCut())%></font></td></tr>
<tr align=center><td colspan=4 style="border:2 solid "><a href="list.jsp" class="linkstyle">继续购买
</a></td>
<td colspan=5 style="border:2 solid "><a href="showcart.jsp?clear=true" class="linkstyle">清空购
物车</a></td>
</tr>
</table>
<%
}else{
if(clear==null||(!clear.equalsIgnoreCase("true"))){
%>
<table align=center><tr><td>您的购物车中暂时没有任何商品，
            <a href = "list.jsp">返回产品列表</a></td></tr></table>
<%
    }
}
%>
</body>
</html>
```

读者意见反馈卡

您购买的书名：__________ 您的姓名：__________ 性 别：□男 □女

年龄：__________ 文化程度：__________ 职 业：__________

邮编：__________ 通信地址：__________ E-mail：__________

您常用的软件：1 __________ 2 __________ 3 __________ 4 __________

您购买本书的原因（可多选）：

□封面与装帧 □引言目录 □正文内容 □丛书风格 □价格 □光盘 □专业性强 □别人介绍

□出版社或作者名声 □售后服务

本书最令您满意的是（可多选）：

□专业性强、覆盖面广 □内容翔实、定位准确 □精益求精、售后服务

您可以承受的图书价格：

□20 元以下 □30 元以下 □40 元以下 □50 元以下 □只要内容好，不论价格

您对本书的评价：

封面装帧：□很好 □较好 □一般 □不满意 建议__________

印刷质量：□很好 □较好 □一般 □不满意 建议__________

正文质量：□很好 □较好 □一般 □不满意 建议__________

写作风格：□很好 □较好 □一般 □不满意 建议__________

专业水平：□很好 □较好 □一般 □不满意 建议__________

您希望增加哪些图书选题：1__________ 2 __________ 3__________

您认为本书有哪些错误：

章______节______页码______行______列______ 图号______ 错误__________应改为 __________

章______节______页码______行______列______ 图号______ 错误__________应改为 __________

章______节______页码______行______列______ 图号______ 错误__________应改为 __________

章______节______页码______行______列______ 图号______ 错误__________应改为 __________

您的其他建议：

1__________

2__________

3__________

请填好本卡后寄给：

清华大学校内金地公司 邮编：100084 电话：（010）62791976-220

《JSP 网络开发技术及整合应用》编辑部收

传真：（010）62788903 公司网址：www.thjd.com.cn E-mail：oyzx_sp@263.net

如需本书可与本编辑部联系邮购，汇款请按以上地址填写，另加邮费 15%（挂号）